

# 15. Data Serialization (ASN.1, XML) and Available Serializable Classes

Created: April 1, 2003  
Updated: March 22, 2004

## The SERIAL API [Library `xserial`:include | src]

The overview for this chapter consists of the following topics:

- Introduction
- Chapter Outline

### Introduction

A SERIAL library is a generic library to provide data serialization in different formats. See also the DATATOOL documentation discussion of generating C++ code for serializable objects from the corresponding ASN.1 definition.

### Chapter Outline

The following is an outline of the topics presented in this chapter:

- Test Cases [src/serial/test]
- CObject[IO]Streams
  - Format Specific Streams: The CObject[IO]Stream classes
  - The CObjectIStream (\*) classes
  - The CObjectOStream (\*) classes
  - The CObjectStreamCopier (\*) classes
  - Type-specific I/O routines
  - The Read hook classes
  - The Write hook classes
  - The Copy hook classes
  - The CObjectHookGuard class
  - Stack Path Hooks
  - The ByteBlock and CharBlock classes

- NCBI C++ Toolkit Network Service (RPC) Clients
  - Introduction and Use
  - Implementation Details
- Verification of Class Member Initialization
  - Initialization Verification in CSerialObject Classes
  - Initialization Verification in Object Streams
- Simplified serialization interface
- The NCBI C++ Toolkit Iterators
  - STL generic iterators
  - CTypeIterator (\*) and CTypeConstIterator (\*)
  - Class hierarchies, embedded objects, and the NCBI C++ type iterators
  - CObjectIterator (\*) and CObjectConstIterator (\*)
  - CStdTypeIterator (\*) and CStdTypeConstIterator (\*)
  - CTypesIterator (\*)
  - Additional Information
- Processing Serial Data
  - Accessing the object header files and serialization libraries
  - Reading and writing serial data
  - Determining Which Header Files to Include
  - Determining Which Libraries to Link To
- User-defined type information
  - Introduction
  - Installing a GetTypeInfo() function: the BEGIN\_/END\_ macros
  - Specifying internal structure and class inheritance: the ADD\_ macros
- Runtime Object Type Information
  - Introduction
  - Motivation

- Object Information Classes
  - CObjectTypeInfo (\*)
  - CConstObjectInfo (\*)
  - CObjectInfo (\*)
- Usage of object type information
- Choice objects in the NCBI C++ Toolkit
  - Introduction
  - C++ choice objects
- Traversing a Data Structure
  - Locating the Class Definitions
  - Accessing and Referencing Data Members
  - Traversing a Biostruc
  - Iterating Over Containers

## Test Cases [\[src/serial/test\]](#)

---

**Available Serializable Classes** (as per NCBI ASN.1 Specifications) [Library `xobjects`: include | src]

The ASN.1 data objects are automatically built from their corresponding specifications in the NCBI ASN.1 data model, using DATATOOL to generate all of the required source code . This set of serializable classes defines an interface to many important sequence and sequence-aware objects that users may directly employ, or extend with their own code. An *Object Manager*(see below) coordinates and simplifies the use of these ASN.1-derived objects.

Serializable Classes

- access [include | src]
- biblio [include | src]
- cdd [include | src]
- cn3d [include | src]
- docsum [include | src]
- entrez2 [include | src]

- featdef [include | src]
- general [include | src]
- id1 [include | src]
- medlars [include | src]
- medline [include | src]
- mim [include | src]
- mla [include | src]
- mmdb1 [include | src]
- mmdb2 [include | src]
- mmdb3 [include | src]
- ncbimime [include | src]
- objprt [include | src]
- proj [include | src]
- pub [include | src]
- pubmed [include | src]
- seq [include | src]
- seqalign [include | src]
- seqblock [include | src]
- seqcode [include | src]
- seqfeat [include | src]
- seqloc [include | src]
- seqres [include | src]
- seqset [include | src]
- submit [include | src]

- taxon1 [include | src]

A Test Application Using the Serializable ASN.1 Classes

- asn2asn [src]

## *CObject[IO]Streams*

---

The following topics are discussed in this section:

- Format Specific Streams: The CObject[IO]Stream classes
- The CObjectIStream (\*) classes
- The CObjectOStream (\*) classes
- The CObjectStreamCopier (\*) classes
- Type-specific I/O routines
- The Read hook classes
- The Write hook classes
- The Copy hook classes
- The CObjectHookGuard class
- Stack Path Hooks
- The ByteBlock and CharBlock classes
- NCBI C++ Toolkit Network Service Clients
- Verification of Class Member Initialization
- Simplified serialization interface

### Format Specific Streams: The *CObject[IO]Stream* classes

The reading and writing of serialized data objects entails satisfying two independent sets of constraints and specifications: (1) *format-specific* parsing and encoding schemes, and (2) *object-specific* internal structures and rules of composition. The NCBI C++ Toolkit implements serial IO processes by combining a set of *object stream* classes with an independently defined set of *data object* classes. These classes are implemented in the *serial* and *objects* directories respectively.

The base classes for the object stream classes are **CObjectIStream** and **CObjectOStream**. Each of these base classes has derived subclasses which specialize in different formats, including XML, binary ASN.1, and text ASN.1. A simple example program, *xml2asn.cpp* (see Box 1), described in Processing serial data, uses these object stream classes in conjunction with a

**CBiostruct** object to translate a file from XML encoding to ASN.1 formats. In this chapter, we consider in more detail the class definitions for object streams, and how the type information associated with the data is used to implement serial input and output.

Each object stream specializes in a serial data format and a direction (in/out). It is not until the input and output operators are applied to these streams, in conjunction with a specified serializable object, that the object-specific type information comes into play. For example, if `instr` is a **CObjectIStream**, the statement: `instr >> myObject` invokes a **Read()** method associated with the input stream, whose sole argument is a **CObjectInfo** for `myObject`.

Similarly, the output operators, when applied to a **CObjectOstream** in conjunction with a serializable object, will invoke a **Write()** method on the output stream which accesses the object's type information. The object's type information defines what tag names and value types should be encountered on the stream, while the **CObject[IO]Stream** subclasses specialize the data serialization format.

The input and output operators (<< and >>) are declared in `serial/serial.hpp` header.

## The **CObjectIStream** classes

**CObjectIStream** is a virtual base class for the **CObjectIStreamXml**, **CObjectIStreamAsn**, and **CObjectIStreamAsnBinary** classes. As such, it has no public constructors, and its user interface includes the following methods:

- **Open()**
- **Close()**
- **GetDataFormat()**
- **ReadFileHeader()**
- **Read()**
- **ReadObject()**
- **ReadSeparateObject()**
- **Skip()**
- **SkipObject()**

There are several **Open()** methods; most of these are static class methods that return a pointer to a newly created **CObjectIStream**. Typically, these methods are used with an **auto\_ptr**, as in:

```
auto_ptr<CObjectIStream> xml_in(CObjectIStream::Open(filename, eSerial_Xml));
```

Here, an XML format is specified by the enumerated value `eSerial_Xml`, defined in ***ESerialDataFormat***. Because these methods are static, they can be used to create a new instance of a ***CObjectIStream*** subclass, and open it with one statement. In this example, a ***CObjectIStreamXml*** is created and opened on the file `filename`.

An additional non-static ***Open()*** method is provided, which can only be invoked as a member function of a previously instantiated object stream (whose format type is of course, implicit to its class). This method takes a ***CNcbiIstream*** and a Boolean argument, specifying whether or not the ***CNcbiIstream*** should also be deleted when the object stream is closed:

```
void Open(CNcbiIstream& inStream, bool deleteInStream = false);
```

The next three methods have the following definitions. ***Close()*** closes the stream.

***GetDataFormat()*** returns the enumerated ***ESerialDataFormat*** for the stream.

***ReadFileHeader()*** reads the first line from the file, and returns it in a string. This might be used for example, in the following context:

```
auto_ptr<CObjectIStream> in(CObjectIStream::Open(fname, eSerial_AsnText));
string type = in.ReadFileHeader();
if (type.compare("Seq-entry") == 0) {
    CSeq_entry seqent;
    in->Read(ObjectInfo(seqent), eNoFileHeader);
    // ...
}
else if (type.compare("Bioseq-set") == 0) {
    CBioseq_set seqset;
    in->Read(ObjectInfo(seqset), eNoFileHeader);
    // ...
}
// ...
```

The ***ReadFileHeader()*** method for the base ***CObjectIStream*** class returns an empty string. Only those stream classes which specialize in ASN.1 text or XML formats have actual implementations for this method.

Several ***Read\*()*** methods are provided for usage in different contexts. ***CObjectIStream::Read()*** should be used for reading a top-level "root" object from a data file. For convenience, the input operator `>>`, as described above, indirectly invokes this method on the input stream, using a ***CObjectTypeInfo*** object derived from `myObject`. By default, the ***Read()*** method first calls ***ReadFileHeader()***, and then calls ***ReadObject()***. Accordingly, calls to ***Read()*** which follow the usage of ***ReadFileHeader()*** must include the optional `eNoFileHeader` argument.

Most data objects also contain embedded objects, and the default behavior of **Read()** is to load the top-level object, along with all of its contained subobjects into memory. In some cases this may require significant memory allocation, and it may be only the top-level object which is needed by the application. The next two methods, **ReadObject()** and **ReadSeparateObject()**, can be used to load subobjects as either persistent data members of the root object or as temporary local objects. In contrast to **Read()**, these methods assume that there is no file header on the stream.

As a result of executing **ReadObject(member)**, the newly created subobject will be instantiated as a member of its parent object. In contrast, **ReadSeparateObject(local)**, instantiates the subobject in the local temporary variable only, and the corresponding data member in the parent object is set to an appropriate *null* representation for that data type. In this case, an attempt to reference that subobject after exiting the scope where it was created generates an error.

The **Skip()** and **SkipObject()** methods allow entire top-level objects and subobjects to be "skipped". In this case the input is still read from the stream and validated, but no object representation for that data is generated. Instead, the data is stored in a delay buffer associated with the object input stream, where it can be accessed as needed. **Skip()** should only be applied to top-level objects. As with the **Read()** method, the optional **ENoFileHeader** argument can be included if the file header has already been extracted from the data stream. **SkipObject(member)** may be applied to subobjects of the root object.

All of the **Read** and **Skip** methods are like wrapper functions, which define what activities take place immediately before and after the data is actually read. How and when the data is then loaded into memory is determined by the object itself. Each of the above methods ultimately calls *objTypeInfo->ReadData()* or *objTypeInfo->SkipData()*, where *objTypeInfo* is the static type information object associated with the data object. This scheme allows the user to install type-specific read, write, and copy hooks, which are described below. For example, the default behavior of loading all subobjects of the top-level object can be modified by installing appropriate read hooks which use the **ReadSeparateObject()** and **SkipObject()** methods where needed.

## The *CObjectOStream* (%20) classes

The output object stream classes mirror the **CObjectIStream** classes. The **CObjectOStream** base class is used to derive the **CObjectOStreamXml**, **CObjectOStreamAsn**, and **CObjectOStreamAsnBinary** classes. There are no public constructors, and the user interface includes the following methods:

- **Open()**
- **Close()**

- ***GetDataFormat()***
- ***WriteFileHeader()***
- ***Write()***
- ***WriteObject()***
- ***WriteSeparateObject()***
- ***Flush()***
- ***FlushBuffer()***

Again, there are several ***Open()*** methods, which are static class methods that return a pointer to a newly created ***CObjectOStream***:

```
static CObjectOStream* Open(const string& fileName, ESerialDataFormat format);
static CObjectOStream* Open(ESerialDataFormat format,
                           const string& fileName, unsigned openFlags = 0);
static CObjectOStream* Open(ESerialDataFormat format,
                           CNcbiOStream& os, bool deleteOutputStream = false);
```

The ***Write\*()*** methods correspond to the ***Read\*()*** methods defined for the input streams. ***Write()*** first calls ***WriteFileHeader()***, and then calls ***WriteObject()***. ***WriteSeparateObject()*** can be used to write a temporary object (and all of its children) to the output stream. It is also possible to install type-specific *write* hooks. Like the ***Read()*** methods, these ***Write()*** methods serve as wrapper functions that define what occurs immediately before and after the data is actually written.

## The *CObjectStreamCopier* classes

The ***CObjectStreamCopier*** class is neither an input nor an output stream class, but a helper class, which allows one to "pass data through" without storing the intermediate objects in memory. Its sole constructor is:

```
CObjectStreamCopier(CObjectIStream& in, CObjectOStream& out);
```

and its most important method is the ***Copy(CObjectTypeInfo&)*** method, which, given an object's description, reads that object from the input stream and writes it to the output stream. The serial formats of both the input and output object streams are implicit, and thus the translation between two different formats is performed automatically.

In keeping with the ***Read*** and ***Write*** methods of the ***CObjectIStream*** and ***CObjectOStream*** classes, the ***Copy*** method takes an optional ***ENoFileHeader*** argument, to indicate that the file header is not present in the input and should not be generated on the output. The ***CopyObject()*** method corresponds to the ***ReadObject()*** and ***WriteObject()*** methods.

As an example, consider how the ***Run()*** method in *xml2asn.cpp* might be implemented differently using the ***CObjectStreamCopier*** class:

```

    int CTestAsn::Run() {
    auto_ptr<CObjectIStream>
    xml_in(CObjectIStream::Open("1001.xml", eSerial_Xml));
    auto_ptr<CObjectOStream>
    txt_out(CObjectOStream::Open("1001.asntxt", eSerial_AsnText));
    CObjectStreamCopier txt_copier(*xml_in, *txt_out);
    txt_copier.Copy(CBiostruc::GetTypeInfo());
    auto_ptr<CObjectOStream>
    bin_out(CObjectOStream::Open("1001.asnbin", eSerial_AsnBinary));
    CObjectStreamCopier bin_copier(*xml_in, *bin_out);
    bin_copier.Copy(CBiostruc::GetTypeInfo());
    return 0;
    }

```

It is also possible to install type-specific **Copy** hooks. Like the **Read** and **Write** methods, the **Copy** methods serve as wrapper functions that define what occurs immediately before and after the data is actually copied.

## Type-specific I/O routines

Much of the functionality needed to read and write serializable objects may be type-specific yet application-driven. Because the specializations may vary with the application, it does not make sense to implement fixed methods, yet we would like to achieve a similar kind of object-specific behavior.

To address these needs, the C++ Toolkit provides hook mechanisms, whereby the needed functionality can be installed with the object's static class type information object. Such hooks can be installed **globally**, where they will be applied on **all** streams where these events occur, or **locally**, where they will only be applied to a selected stream.

For any given object and specific stream, at most one read hook and one write hook is "active". If `myObject` has a locally installed read hook as well as a global read hook, then the locally installed hook will override the global hook when a read occurs on the "local" stream. Read events on all of the other "non-local" streams will of course, trigger the globally installed hook. Designating multiple read/write hooks (both local and global) for a selected object does not generate an error. Older or less specific hooks are simply overridden by the more specific or most recently installed hook.

## The *Read* hook classes

All of the different contexts in which an object might be encountered on an input stream can be reduced to three cases:

1. as a stand-alone object
2. as a data member of a containing object
3. as a variant of a *choice* object

Hooks can be installed for each of these contexts, depending on the desired level of specificity. Corresponding to these contexts, three abstract base classes provide the foundations for deriving new *Read* hooks:

- ***CReadObjectHook***
- ***CReadClassMemberHook***
- ***CReadChoiceVariantHook***

Each of these base hook classes exists only to define a pure virtual ***Read*** method, which can then be implemented (in a derived subclass) to install the desired type of read hook. If the goal is to apply the new ***Read*** method in all contexts, then the new hook should be derived from the ***CReadObjectHook*** class, and registered with the object's static type information object. For example, to install a new ***CReadObjectHook*** for a ***CBioseq***, one might use:

```
CObjectTypeInfo(CBioseq::GetTypeInfo()).SetGlobalReadHook(myReadBioseqHook);
```

Another way of installing hooks of any type (read/write/copy, object/member/variant) is provided by ***CObjectHookGuard*** class described below.

Alternatively, if the desired behavior is to trigger the specialized ***Read*** method only when the object occurs as a data member of a particular containing class, then the new hook should be derived from the ***CReadClassMemberHook***, and registered with that member's type information object:

```
CObjectTypeInfo(CBioseq::GetTypeInfo()).FindMember("Seq-inst").SetGlobalReadHook(myReadSeqinstHook);
```

Similarly, one can install a read hook that will only be triggered when the object occurs as a choice variant:

```
CObjectTypeInfo(CSeq_entry::GetTypeInfo()).FindVariant("Bioseq").SetGlobalReadHook(myReadBioseqHook);
```

The new hook classes for these examples should be derived from ***CReadObjectHook***, ***CReadClassMemberHook***, and ***CReadChoiceVariantHook***, respectively. In the first case, all occurrences of ***CBioseq*** on any input stream will trigger the new ***Read*** method. In contrast, the last case installs this new ***Read*** method to be triggered only when the ***CBioseq*** occurs as a choice variant in a ***CSeq\_entry*** object.

All of the virtual ***Read*** methods take two arguments: a ***CObjectIStream*** and a reference to a ***CObjectInfo***. For example, the ***CReadObjectHook*** class declares the ***ReadObject()*** method as:

```
virtual void ReadObject(CObjectIStream& in,
                      const CObjectInfo& object) = 0;
```

The ***ReadClassMember*** and ***ReadChoiceVariant*** hooks differ from the ***ReadObject*** hook class, in that the second argument to the virtual ***Read*** method is an iterator, pointing to the object type information for a data member or choice variant respectively.

In summary, to install a read hook for an object type:

1. derive a new class from the appropriate hook class:
  - if the target object occurs in any context, use the **CReadObjectHook** class.
  - if the target object occurs as a data member, use the **CReadClassMemberHook** class.
  - if the target object occurs as a choice variant, use the **CReadChoiceVariant Hook** class.
2. implement the virtual **Read** method for the new class.
3. install the hook, using the **SetGlobalReadHook()** or **SetLocalReadHook()** method defined in
  - **CObjectTypeInfo** for a **CReadObjectHook**
  - **CMemberInfo** for a **CReadClassMemberHook**
  - **CVariantInfo** for a **CReadChoiceVariantHook**

or use **CObjectHookGuard** class to install any of these hooks.

In many cases you will need to read the hooked object and do some special processing, or to skip the entire object. To simplify object reading or skipping all base hook classes have **DefaultRead()** and **DefaultSkip()** methods taking the same arguments as the user provided **ReadXXXX()** methods. Thus, to read a bioseq object from a hook:

```
void CMyReadObjectHook::ReadObject(CObjectIStream& in, const CObjectInfo& object)
{
    DefaultRead(in, object);
    // Do some user-defined processing of the bioseq
}
```

Note that from a choice variant hook you can not skip stream data -- this could leave the choice object in an uninitialized state. For this reason the **CReadChoiceVariantHook** class has no **DefaultSkip()** method.

For a good example of using a **CReadClassMemberHook** object, see the *asn2asn.cpp* and *testserial.cpp* demo programs.

## The Write hook classes

The *Write* hook classes parallel the *Read* hook classes, and again, we have three base classes:

- **CWriteObjectHook**
- **CWriteClassMemberHook**

- **CWriteChoiceVariantHook**

These classes define the pure virtual methods:

```
CWriteObjectHook::WriteObject(CObjectOStream&,
                               const CConstObjectInfo& object) = 0;

CWriteClassMemberHook::WriteClassMember(CObjectOStream&,
                                         const CConstObjectInfoMI& member) = 0;

CWriteChoiceVariantHook::WriteChoiceVariant(CObjectOStream&,
                                             const CConstObjectInfoCV& variant) = 0;
```

Like the read hooks, your derived write hooks can be installed by invoking the **SetGlobalWriteObjectHook()** or **SetLocalWriteObjectHook()** methods for the appropriate type information objects. Corresponding to the examples for read hooks then, we would have:

```
CObjectTypeInfo(CBioseq::GetTypeInfo()).SetGlobalWriteHook(myWriteBioseqHook);
CObjectTypeInfo(CBioseq::GetTypeInfo()).FindMember("Seq-inst").SetGlobalWriteHook
(myWriteSeqinstHook);

CObjectTypeInfo(CSeq_entry::GetTypeInfo()).FindVariant("Bioseq").SetGlobalWriteHook
(myWriteBioseqHook);
```

**CObjectHookGuard** class provides is a simple way to install write hooks.

The *asn2asn.cpp* and *testserial.cpp* demo programs also demonstrate the usage of the **CWriteClassMemberHook** class.

## The Copy hook classes

As with the *Read* and *Write* hook classes, there are three base classes which define the following *Copy* methods:

```
CCopyObjectHook::CopyObject(CObjectStreamCopier& copier,
                             const CObjectTypeInfo& object) = 0;

CCopyClassMemberHook::CopyClassMember(CObjectStreamCopier& copier,
                                       const CObjectTypeInfoMI& member) = 0;

CCopyChoiceVariantHook::CopyChoiceVariant(CObjectStreamCopier&,
                                           const CObjectTypeInfoCV& variant) = 0;
```

Newly derived copy hooks can be installed by invoking the **SetGlobalCopyObjectHook()** or **SetLocalCopyObjectHook()** methods for the appropriate type information objects. The other way of installing hooks is described below in the **CObjectHookGuard** section.

To do default copying of an object in the overloaded hook method each of the base copy hook classes has **DefaultCopy()** method.

## The *CObjectHookGuard* class

To simplify hooks usage ***CObjectHookGuard*** class may be used. It's a template class: the template parameter is the class to be hooked (in case of member or choice variant hooks it's the parent class of the member).

The *CObjectHookGuard* class has several constructors for installing different hook types. The last argument to all constructors is a stream pointer. By default the pointer is NULL and the hook is intalled as a global one. To make the hook stream-local pass the stream to the guard constructor.

- Object read/write hooks:

```
CObjectHookGuard(CReadObjectHook& hook, CObjectIStream* in = 0);
CObjectHookGuard(CWriteObjectHook& hook, CObjectOStream* out = 0);
```

- Class member read/write hooks:

```
CObjectHookGuard(string id, CReadClassMemberHook& hook, CObjectIStream* in = 0);
CObjectHookGuard(string id, CWriteClassMemberHook& hook, CObjectOStream* out = 0);
```

The string "id" argument is the name of the member in ASN.1 specification for generated classes.

- Choice variant read/write hooks:

```
CObjectHookGuard(string id, CReadChoiceVariantHook& hook, CObjectIStream* in = 0);
CObjectHookGuard(string id, CWriteChoiceVariantHook& hook, CObjectOStream* out = 0);
```

The string "id" argument is the name of the variant in ASN.1 specification for generated classes.

The guard's destructor will uninstall the hook. Since all hook classes are derived from *CObject* and stored as *CRef<>*-s, the hooks are destroyed automatically when uninstalled. For this reason it's recommended to create hook objects on heap.

## Stack Path Hooks

When using serialization hooks one might want to specify a more specific context when such hook should be triggered. For example, "I want to hook the reading of object A when and only when it is a member of object B, not all occurrences of object A", or "I want to hook the reading of all members named 'Title' in all objects, not only in a specific one". The serial library makes it possible to set serialization hooks by string that describes a place (or stack path), for example:

*TypeName.Member1.Member2.HookedMember*

The format of the string is as follows:

```
Stackpath ::= (TypeName | Wildcard) ('.' (MemberName | Wildcard))+
```

Where *TypeName* and *MemberName* are strings, '.' is a separator. *Wildcard* is defined as

```
Wildcard ::= ('?' | '*')
```

Here the question mark means "one member with any name", while the asterisk means "one or more members with any names".

As with regular serialization hooks, it is possible to install a path hook for a specific object:

```
CObjectTypeInfo(CBioseq::GetTypeInfo()).SetPathReadHook(in, path, myReadBioseqHook);
```

a data member of an object:

```
CObjectTypeInfo(CBioseq::GetTypeInfo()).FindMember("Seq-nst").SetPathReadHook(in,
path, myReadSeqinstHook);
```

or a variant of a choice object:

```
CObjectTypeInfo(CSeq_entry::GetTypeInfo()).FindVariant("Bioseq").
SetPathReadHook(in, path, myReadBioseqHook);
```

Here `in` is a pointer to an input object stream. If it is equal to zero, the hook will be installed globally, otherwise - for that particular stream. In addition to that, it is possible to install such hooks in object streams. So, for example to install a read hook on all string data members and choice variants named `LastName`, one could use either the following code:

```
CObjectTypeInfo(CStdTypeInfo<string>::GetTypeInfo()).SetPathReadHook(in, "*.LastName",
myObjHook);
```

Or this one:

```
in->SetPathReadObjectHook("*.LastName", myObjHook);
```

Setting path hooks directly in streams also makes it possible to differentiate between `LastName` being a data member and choice variant. So, for example

```
in->SetPathReadMemberHook("*.LastName", myMemHook);
```

will catch all data members and skip choice variants; while

```
in->SetPathReadVariantHook("*.LastName", myVarHook);
```

will trigger for all variants and skip data members.

## Stream Iterators

When working with a stream, it is sometimes convenient to be able to read or write data elements directly, bypassing the standard data storage mechanism. For example, when reading a large container object, the purpose could be to process its elements. It is possible to read everything at once, but this could require a lot of memory to store the data in. An alternative approach, which greatly reduces the amount of required memory, could be to read elements one by one, process them as they arrive, and then discard. Or, when writing a container, one could construct it in memory only partially, and then add missing elements 'on the fly' - where appropriate. To make it possible, the SERIAL library introduces *stream iterators*. Needless to say, the most convenient way of using this mechanism is in read/write hooks.

SERIAL library defines the following stream iterator classes: ***CStreamClassMemberIterator*** and ***CStreamContainerIterator*** for input streams, and ***CStreamClassMember*** and ***CStreamContainer*** for output ones.

Reading a container could look like this:

```

    for ( CStreamContainerIterator i(in, containerType); i; ++i ) {
        CElementClass element;
        i >> element;
    }

```

Writing - like this:

```

COutputStream o(out, containerType);
// set<CElementClass> container - is your own data container defined elsewhere
for ( set<CElementClass>::const_iterator i = container.begin(); i != container.end(); +
+i ) {
    const CElementClass& element = *i;
    o << element;
}

```

For more examples of using stream iterators please refer to `asn2asn` sample application.

## The *ByteBlock* and *CharBlock* classes

**CObject[IO]Stream::ByteBlock** class may be used for non-standard processing of an OCTET STRING data, e.g. from a read/write hooks. The **CObject[IO]Stream::CharBlock** class has almost the same functionality, but may be used for VisibleString data processing.

An example of using ByteBlock or CharBlock classes is generating data on-the-fly in a write hook. To use block classes:

1. Initialize the block variable with an i/o stream and, in case of output stream, the length of the block.
2. Use Read()/Write() functions to process block data
3. Close the block with the End() function

Below is an example of using **CObjectOStream::ByteBlock** in an object write hook for non-standard data processing. Note, that ByteBlock and CharBlock classes read/write data only. You should also provide some code for writing class' and members' tags.

Since OCTET STRING and VisibleString in the NCBI C++ Toolkit are implemented as **vector<char>** and **string** classes, which have no serialization type info, you can not install a read or write hook for these classes. The example also demonstrates how to process members of these types using the containing class hook. Another example of using CharBlock with write hooks can be found in **testserial.cpp** application.

```

void CWriteMyObjectHook::WriteObject(CObjectOStream& out,
                                     const CConstObjectInfo& object)
{
    const CMyObject& obj = *reinterpret_cast<const CMyObject*>
        (object.GetObjectPtr());
    if ( NothingToProcess(obj) ) {
        // No special processing - use default write method
        DefaultWrite(out, object);
        return;
    }
    // Write object open tag

```

```

out.BeginClass(object.GetClassTypeInfo());
// Iterate object members
for (CConstObjectInfo::CMemberIterator member =
    object.BeginMembers(); member; ++member) {
    if ( NeedProcessing(member) ) {
        // Write the special member manually
        out.BeginClassMember(member.GetMemberInfo()->GetId());
        // Start byte block, specify output stream and block size
        size_t length = GetRealDataLength(member);
        CObjectOStream::ByteBlock bb(out, length);
        // Processing and output
        for (int i = 0; i < length; ) {
            char* buf;
            int buf_size;
            // Assuming ProcessData() generates the data from "member",
            // starting from position "i" and stores the data to "buf"
            ProcessData(member, i, &buf_size, &buf);
            i += buf_size;
            bb.Write(buf, buf_size);
        }
    }
    // Close the byte block
    bb.End();
    // Close the member
    out.EndClassMember();
}
else {
    // Default writer for members without special processing
    if ( member.IsSet() )
        out.WriteClassMember(member);
}
// Close the object
out.EndClass();
}

```

## NCBI C++ Toolkit Network Service (RPC) Clients

The following topics are discussed in this section:

- Introduction and Use
- Implementation Details

### Introduction and Use

The C++ Toolkit now contains `datatool`-generated classes for certain ASN.1-based network services: at the time of this writing, *Entrez2*, *ID1*, and *MedArch*. (There is also an independently written class for the *Taxon1* service, **CTaxon1**, which this page does not discuss further.) All of these classes, declared in headers named *objects/.../client(\_).hpp*, inherit certain useful properties from the base template **CRPCClient<>**:

- They normally defer connection until the first actual query, and disconnect automatically when destroyed, but let users request either action explicitly.
- They are designed to be thread-safe (but, at least for now, maintain only a single connection per instance, so forming pools may be appropriate).

The usual interface to these classes is through a family of methods named **AskXxx**, each of which takes a request of an appropriate type and an optional pointer to an object that will receive the full reply and returns the corresponding reply choice. For example, **CEntrez2Client::AskEval\_boolean** takes a request of type **const CEntrez2\_eval\_boolean&** and an optional pointer of type **CEntrez2\_reply\***, and returns a reply of type **CRef<CEntrez2\_boolean\_reply>**. All of these methods automatically detect server-reported errors or unexpected reply choices, and throw appropriate exceptions when they occur. There are also lower-level methods simply named **Ask**, which may come in handy if you do not know what kind of query you will need to make.

In addition to these standard methods, there are certain class-specific methods: **CEntrez2Client** adds **GetDefaultRequest** and **SetDefaultRequest** for dealing with those fields of **Entrez2-request** besides `request` itself, and **CID1Client** adds **{Get,Set}AllowDeadEntries** (off by default) to control how to handle the result choice `gotdeadsequery`.

## Implementation Details

In order to get `datatool` to generate classes for a service, you must add some settings to the corresponding `modulename.def` file. Specifically, you must set `[-]clients` to the relevant base file name (typically `service_client`), and add a correspondingly named section containing the entries listed in Table 1. (If a single specification defines multiple protocols for which you would like `datatool` to generate classes, you may list multiple client names, separated by spaces.)

**Table 1. Network Service Client Generation Parameters**

Name	Value
<i>class</i> (REQUIRED)	C++ class name to use.
<i>service</i>	Named service to connect to; if you do not define this, you will need to override <b>x_Connect</b> in the user class.
<i>serialformat</i>	Serialization format: normally <i>AsnBinary</i> , but <i>AsnText</i> and <i>Xml</i> are also legal.
<i>request</i> (REQUIRED)	ASN.1 type for requests; may include a module name, a field name (as with <i>Entrez2</i> ), or both. Must be a CHOICE.
<i>reply</i> (REQUIRED)	ASN.1 type for replies, as above.
<i>reply.choice_name</i>	Reply choice appropriate for requests of type <code>choice_name</code> ; defaults to <code>choice_name</code> as well, and determines the return type of <b>AskChoice_name</b> .

Name	Value
	May be set to <i>special</i> to suppress automatic method generation and let the user class handle the whole thing.

## Verification of Class Member Initialization

When serializing an object, it is important to verify that all mandatory primitive data members (e.g. strings, integers) are given a value. The NCBI C++ Toolkit implements this through a data initialization verification mechanism. In this mechanism, the value itself is not validated; that is, it still could be semantically incorrect. The purpose of the verification is only to make sure that the member has been assigned some value. The verification also provides for a possibility to check whether the object data member has been initialized or not. This could be useful when constructing such objects in memory.

From this perspective, each data member (XXX) of a serial object generated by DATATOOL from an ASN or XML specification has the **IsSetXXX()** and **CanGetXXX()** methods. Also, input and output streams have **SetVerifyData()** and **GetVerifyData()** methods. The purpose of **CanGetXXX()** method is to answer the question whether it is safe or not to call the corresponding **GetXXX()**. The meaning of **IsSetXXX()** is whether the data member has been assigned a value explicitly (using assignment function call, or as a result of reading from a stream) or not. The stream's **SetVerifyData()** method defines a stream behavior in case it comes across an uninitialized data member.

There are three kinds of object data members:

- optional ones,
- mandatory with a default value,
- mandatory with no default value.

Optional members and mandatory ones with no default have "no value" initially. As such, they are "ungetatable"; that is, **GetXXX()** throws an exception (this is also configurable though). Mandatory members with a default are always gettable, but not always set. It is possible to assign a default value to a mandatory member with a default value. In this case it becomes set, and as such will be written into an output stream.

The discussion above refers only to primitive data members, such as strings, or integers. The behavior of containers is somewhat different. All containers are pre-created on the parent object construction, so for container data members **CanGetXXX()** always returns TRUE. This can be justified by the fact that containers have a sort of "natural default value" - empty. Also, **IsSetXXX()** will return TRUE if the container is either mandatory, or has been read (even if empty) from the input stream, or **SetXXX()** was called for it.

The following additional topics are discussed in this section:

- Initialization Verification in CSerialObject Classes
- Initialization Verification in Object Streams

## Initialization Verification in CSerialObject Classes

**CSerialObject** defines two functions to manage how uninitialized data members would be treated:

```
static void SetVerifyDataThread(ESerialVerifyData verify);
static void SetVerifyDataGlobal(ESerialVerifyData verify);
```

The **SetVerifyDataThread()** defines the behavior of **GetXXX()** for the current thread, while the **SetVerifyDataGlobal()** for the current process. Please note, that disabling **CUnassigned-Member** exceptions in **GetXXX()** function is potentially dangerous because it could silently return garbage.

The behavior of initialization verification has been designed to allow for maximum flexibility. It is possible to define it using environment variables, and then override it in a program, and vice versa. It is also possible to force a specific behavior, no matter what the program sets, or could set later on. The **ESerialVerifyData** enumerator could have the following values:

- *eSerialVerifyData\_Default*
- *eSerialVerifyData\_No*
- *eSerialVerifyData\_Never*
- *eSerialVerifyData\_Yes*
- *eSerialVerifyData\_Always*

Setting **eSerialVerifyData\_Never** or **eSerialVerifyData\_Always** results in a "forced" behavior: setting **eSerialVerifyData\_Never** prohibits later attempts to enable verification; setting **eSerialVerifyData\_Always** prohibits attempts to disable it. The default behavior could be defined from the outside, using the SET\_VERIFY\_DATA\_GET environment variable:

```
SET_VERIFY_DATA_GET ::= ( 'NO' | 'NEVER' | 'YES' | 'ALWAYS' )
```

Alternatively, the default behavior can also be set from a program code using **CSerialObject::SetVerifyDataXXX()** functions.

Setting the environment variable to "Never/Always" overrides any attempt to change the verification behavior in the program. Setting "Never/Always" for the process overrides attempts to change it for a thread. "Yes/No" setting is less restrictive: the environment variable, if present, provides the default, which could then be overridden in a program, or thread. Here thread settings supersede the process ones.

## Initialization Verification in Object Streams

Data member verification in object streams is a bit more complex.

First, it is possible to set the verification behavior on three different levels:

- for a specific stream (**SetVerifyData()**),
- for all streams created by a current thread (**SetVerifyDataThread()**),
- for all stream created by the current process (**SetVerifyDataGlobal()**).

Second, there are more options in defining what to do in case of an uninitialized data member:

- throw an exception;
- skip it on writing (write nothing), and leave uninitialized (as is) on reading;
- write some default value on writing, and assign it on reading (even though there is no default).

So, **ESerialVerifyData** enumerator could now have two more values: **eSerialVerifyData\_DefValue** and **eSerialVerifyData\_DefValueAlways**. In this case, on reading a missing data member, stream initializes it with a "default" (usually 0); on writing the unset data member, it writes it "as is". For comparison: in the "No/Never" case on reading a missing member stream could initialize it with a "garbage", while on writing it writes nothing. The latter case produces semantically incorrect output, but preserves information of what has been set, and what is not set.

The default behavior could be set similarly to CSerialObject. The environment variables are as follows:

```
SET_VERIFY_DATA_READ  ::= ( 'NO' | 'NEVER' | 'YES' | 'ALWAYS' | 'DEFVALUE' | 'DEFVALUE_ALWAYS' )
SET_VERIFY_DATA_WRITE ::= ( 'NO' | 'NEVER' | 'YES' | 'ALWAYS' | 'DEFVALUE' | 'DEFVALUE_ALWAYS' )
```

## Simplified Serialization Interface

The reading and writing of serial object requires creation of special object streams which encode and decode data. While such streams provide with a greater flexibility in setting the formatting parameters, in some cases it is not needed - the default behavior is quite enough. NCBI C++ toolkit library makes it possible to use the standard I/O streams in this case, thus hiding the creation of object streams. So, the serialization would look like this:

```
cout << MSerial_AsnText << obj;
```

The only information that is always needed is the output format. It is defined by the following stream manipulators:

- **MSerial\_AsnText**
- **MSerial\_AsnBinary**
- **MSerial\_Xml**

Few additional manipulators define the handling of un-initialized object data members:

- **MSerial\_VerifyDefault**
- **MSerial\_VerifyNo**

- *MSerial\_VerifyYes*
- *MSerial\_VerifyDefValue*

## The NCBI C++ Toolkit Iterators

---

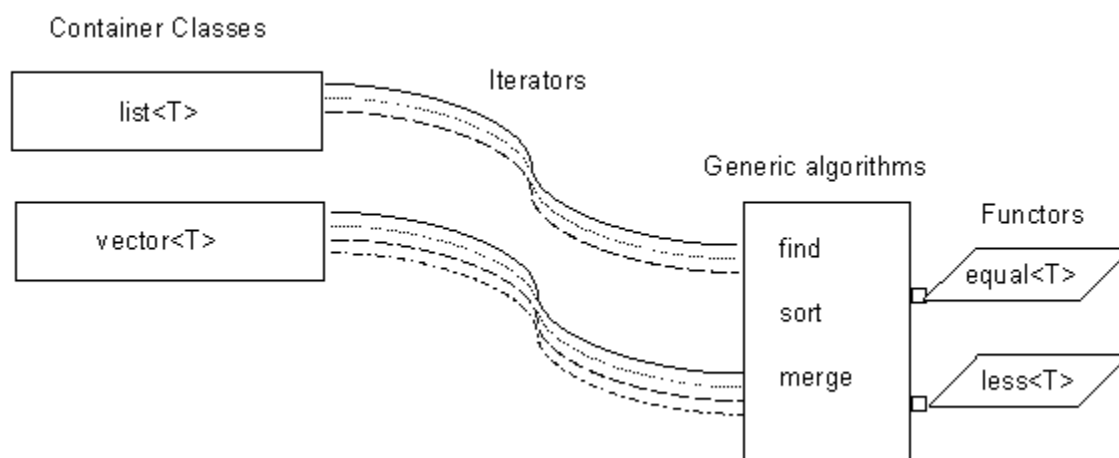
The following topics are discussed in this section:

- STL generic iterators
- CTypeIterator (\*) and CTypeConstIterator (\*)
- Class hierarchies, embedded objects, and the NCBI C++ type iterators
- CObjectIterator (\*) and CObjectConstIterator (\*)
- CStdTypeIterator (\*) and CStdTypeConstIterator (\*)
- CTypesIterator (\*)
- Additional Information

### STL generic iterators

Iterators are an important cornerstone in the generic programming paradigm - they serve as intermediaries between generic containers and generic algorithms. Different containers have different access properties, and the interface to a generic algorithm must account for this. This is depicted graphically below, for the **list** and **vector** containers and the **sort**, **find**, and **merge** algorithms.

See Figure 1.



**Figure 1:** Iterators for some STL classes

The **vector** class allows *input*, *output*, *bidirectional*, and *random access* iterators. In contrast, the **list** container class does **not** allow random access to its elements. This is depicted graphically by one less strand in the ribbon connector. In addition to the iterators, the generic algorithms may require function objects such as *less<T>* to support the template implementations.

The STL standard iterators are designed to iterate through any STL container of homogeneous elements, e.g., *vectors*, *lists*, *deque*s, *stack*s, *maps*, *multimaps*, *sets*, *multisets*, etc. A prerequisite however, is that the container must have **begin()** and **end()** functions defined on it as start and end points for the iteration.

But while these standard iterators are powerful tools for generic programming, they are of no help in iterating over the elements of *aggregate* objects - e.g., over the heterogeneous data members of a class object. As this is an essential operation in processing serialized data structures, the NCBI C++ Toolkit provides additional types of iterators for just this purpose. In the section on Runtime object type information, we described the **CMemberIterator** and **CVariantIterator** classes, which provide access to the instance and type information for **all** of the data members and choice variants of a class or choice object. In some cases however, we may wish to visit only those data members which are of a certain type, and do not require any type information. The iterators described in this section are of this type.

## *CTypeIterator* (%20) and *CTypeConstIterator* (%20)

The **CTypeIterator** and **CTypeConstIterator** can be used to traverse a structured object, stopping at all data members of a specified type. For example, it is very common to represent a linked list of objects by encoding a next field that embeds an object of the same type. One way to traverse the linked list then, would be to "iterate" over all objects of that type, beginning at the head of the list. For example, suppose you have a **CPerson** class defined as:

```
class CPerson
{
public:
    CPerson(void);
    CPerson(const string& name, const string& address, CPerson* p);
    virtual ~CPerson(void);
    static const CTypeInfo* GetTypeInfo(void);
    string m_Name, m_Addr;
    CPerson *m_NextDoor;
};
```

Given this definition, one might then define a neighborhood using a single **CPerson**. Assuming a function **FullerBrushMan(CPerson&)** must now be applied to each person in the neighborhood, this could be implemented using a **CTypeIterator** as follows:

```
CPerson neighborhood("Moe", "123 Main St",
    new CPerson("Larry", "127 Main St",
    new CPerson("Curly", "131 Main St", 0)));
```

```
for (CTypeIterator<CPerson> house(Begin(neighborhood)); house; ++house ) {
    FullerBrushMan(*house);
}
```

In this example, the data members visited by the iterator are of the same type as the top-level aggregate object, since `neighbor` is an instance of **CPerson**. Thus, the first "member" visited is the top-level object itself. This is not always the case however. The top-level object is only included in the iteration when it is an instance of the type specified in the template argument (**CPerson** in this case).

All of the NCBI C++ Toolkit type iterators are *recursive*. Thus, since `neighborhood` has **CPerson** data members, which in turn contain objects of type **CPerson**, all of the nested data members will also be visited by the above iterator. More generally, given a hierarchically structured object containing data elements of a given type nested several levels deep, the NCBI C++ Toolkit type iterators effectively generate a "flat" list of all these elements.

It is not difficult to imagine situations where recursive iterators such as the **CTypeIterator** could lead to infinite loops. An obvious example of this would be a doubly-linked list. For example, suppose **CPerson** had both `previous` and `next` data members, where `x->next->previous == x`. In this case, visiting `x` followed by `x->next` would lead back to `x` with no terminating condition. To address this issue, the **Begin()** function accepts an optional second argument, **eDetectLoops**. `eDetectLoops` is an *enum* value which, if included, specifies that the iterator should detect and avoid infinite loops. The resulting iterator will be somewhat slower but can be safely used on objects whose references might create loops.

Let's compare the syntax of this new iterator class to the standard iterators:

```
ContainerType<T> x;
for (ContainerType<T>::IteratorType i = x.begin(); i != x.end(); ++i)
for (CTypeIterator<T> i(Begin(ObjectName)); i; ++i)
```

The standard iterator begins by pointing to the first item in the container `x.begin()`, and with each iteration, visits subsequent items until the end of the container `x.end()` is reached. Similarly, the **CTypeIterator** begins by pointing to the first data member of `ObjectName` that is of type **T**, and with each iteration, visits subsequent data members of type **T** until the end of the top-level object is reached.

A lot of code actually uses `= Begin(...)` instead of `(Begin(...))` to initialize iterators; although the alternate syntax is somewhat more readable and often works, some compilers can mis-handle it and give you link errors. As such, direct initialization as shown above generally works better. Also, note that this issue only applies to construction; you should (and must) continue to use `=` to reset existing iterators.

How are generic iterators such as these implemented? The **Begin()** expression returns an object containing a pointer to the input object `ObjectName`, as well as a pointer to a **CTypeInfo** object containing *type information* about that object. On each iteration, the `++` operator examines the **current** type information to find the next data member which is of type **T**. The current object, its type information, and the state of iteration is pushed onto a local stack, and the iterator is then reset with a pointer to the next object found, and in turn, a pointer to its type information. Each

data member of type **T** (or derived from type **T**) must be capable of providing its own type information as needed. This allows the iterator to recursively visit all data members of the specified type at all levels of nesting.

More specifically, each object included in the iteration, as well as the initial argument to **Begin()**, must have a statically implemented **GetTypeInfo()** class member function to provide the needed type information. For example, all of the serializable objects generated by `datatool` in the `src/objects` subtrees have **GetTypeInfo()** member functions. In order to apply type iterators to user-defined classes (as in the above example), these classes must also make their type information explicit. A set of macros described in the section on *User-defined Type Information* are provided to simplify the implementation of the **GetTypeInfo()** methods for user-defined classes. The example included at the end of this section (see Additional Information) uses several of the C++ Toolkit type iterators and demonstrates how to apply some of these macros.

The **CTypeConstIterator** parallels the **CTypeIterator**, and is intended for use with *const* objects (i.e. when you want to prohibit modifications to the objects you are iterating over). For *const* iterators, the **ConstBegin()** function should be used in place of **Begin()**.

## Class hierarchies, embedded objects, and the NCBI C++ type iterators

As emphasized above, all of the objects visited by an iterator must have the **GetTypeInfo()** member function defined in order for the iterators to work properly. For an iterator that visits objects of type **T**, the type information provided by **GetTypeInfo()** is used to identify:

- data members of type **T**
- data members containing objects of type **T**
- data members derived from type **T**
- data members containing objects derived from type **T**

Explicit encoding of the class hierarchy via the **GetTypeInfo()** methods allows the user to deploy a type iterator over a single specified type which may in practice include a set of types via inheritance. The section Additional Information details a simple example of this feature. The pre-processor macros used in this example which support the encoding of hierarchical class relations are described in the *User-defined Type Information* section. A further generalization of this idea is implemented by the **CTypesIterator** described later.

## CObjectIterator and CObjectConstIterator

Because the **CObject** class is so central to the Toolkit, a special iterator is also defined, which can automatically distinguish **CObjects** from other class types. The syntax of a **CObjectIterator** is:

```
for (CObjectIterator i(Begin(ObjectName)); i; ++i)
```

Note that there is no need to specify the object type to iterate over, as the type **CObject** is built into the iterator itself. This iterator will recursively visit all **CObjects** contained or referenced in `ObjectName`. The **CObjectConstIterator** is identical to the **CObjectIterator** but is designed to operate on *const* elements and uses the **ConstBegin()** function.

User-defined classes that are derived from **CObject** can also be iterated over (assuming their **GetTypeInfo()** methods have been implemented). In general however, care should be used in applying this type of iterator, as not all of the NCBI C++ Toolkit classes derived from **CObject** have implementations of the **GetTypeInfo()** method. **All** of the generated serializable objects in *include/objectsdo* have a defined **GetTypeInfo()** member function however, and thus can be iterated over using either a **CObjectIterator** or a **CTypeIterator** with an appropriate template argument.

## CStdTypeIterator (%20) and CStdTypeConstIterator (%20)

All of the type iterators described thus far require that each object visited must provide its own type information. Hence, none of these can be applied to standard types such as *int*, *float*, *double* or the STL type *string*. The **CStdTypeIterator** and **CStdTypeConstIterator** classes selectively iterate over data members of a specified type. But for these iterators, the type **must** be a simple C type (*int*, *double*, *char\**, etc.) or an STL type *string*. For example, to iterate over all the *string* data members in a **CPerson** object, we could use:

```
for (CStdTypeIterator<string> i(Begin(neighborhood)); i; ++i) {
    cout << *i << ' ';
}
```

The **CStdTypeConstIterator** is identical to the **CStdTypeIterator** but is designed to operate on *const* elements and requires the **ConstBegin()** function.

Code examples using the **CTypeIterator** and **CStdTypeIterator** are given in *ctypeiter.cpp* (see Box 2; for *ctypeiter.hpp*, see Box 3).

## CTypesIterator (%20)

Sometimes it is necessary to iterate over a set of types contained inside an object. The **CTypeIterator**, as its name suggests, is designed for this purpose. For example, suppose you have loaded a gene sequence into memory as a **CBioseq** (named *seq*), and want to iterate over all of its references to genes and organisms. The following sequence of statements defines an iterator that will step through all of *seq*'s data members (recursively), stopping only at references to gene and organism citations:

```
CTypesIterator i;
CType<CGene_ref>::AddTo(i);           // define the types to stop at
CType<COrg_ref>::AddTo(i);

for (i = Begin(seq); i; ++i) {

    if (CType<CGene_ref>::Match(i)) {
        CGene_ref* geneRef = CType<CGene_ref>::Get(i);
        ...
    }
}
```

```

    }
    else if (CType<COrg_ref>::Match(i) {
        COrg_ref* orgRef = CType<COrg_ref>::Get(i);
        ...
    }
}

```

Here, **CType** is a helper template class that simplifies the syntax required to use the multiple types iterator:

- **CType<TypeName>::AddTo(i)** specifies that iterator *i* should stop at type **TypeName**.
- **CType<TypeName>::Match(i)** returns *true* if the specified type **TypeName** is the type currently pointed to by iterator *i*.
- **CType<TypeName>::Get(i)** retrieves the object currently pointed to by iterator *i* if there is a type match to **TypeName**, and otherwise returns 0. In the event there is a type match, the retrieved object is type cast to **TypeName** before it is returned.

The **Begin()** expression is as described for the above **CTypeIterator** and **CTypeConstIterator** classes. The **CTypeConstIterator** is the *const* implementation of this type of iterator, and requires the **ConstBegin()** function.

## Additional Information

The following example demonstrates how the class hierarchy determines which data members will be included in a type iterator. The example uses five simple classes:

- Class **CA** contains a single *int* data member and is used as a target object type for the type iterators demonstrated.
- class **CB** contains an *auto\_ptr* to a **CA** object.
- Class **CC** is derived from **CA** and is used to demonstrate the usage of class hierarchy information.
- Class **CD** contains an *auto\_ptr* to a **CC** object, and, since it is derived from **CObject**, can be used as the object pointed to by a **CRef**.
- Class **CX** contains both pointers-to and instances-of **CA**, **CB**, **CC**, and **CD** objects, and is used as the argument to **Begin()** for the demonstrated type iterators.

The preprocessor macros used in this example implement the **GetTypeInfo()** methods for the classes, and are described in the section on *User-defined type information*.

```

// Define a simple class to use as iterator's target objects
class CA
{
public:
    CA() : m_Data(0) {};

```

```

        CA(int n) : m_Data(n) {};
        static const CTypeInfo* GetTypeInfo(void);
        int m_Data;
    };
    // Define a class containing an auto_ptr to the target class
    class CB
    {
    public:
        CB() : m_a(0) {};
        static const CTypeInfo* GetTypeInfo(void);
        auto_ptr<CA> m_a;
    };
    // define a subclass of the target class
    class CC : public CA
    {
    public:
        CC() : CA(0){};
        CC(int n) : CA(n){};
        static const CTypeInfo* GetTypeInfo(void);
    };

    // define a class derived from CObject to use in a CRef
    // this class also contains an auto_ptr to the target class
    class CD : public CObject
    {
    public:
        CD() : m_c(0) {};
        static const CTypeInfo* GetTypeInfo(void);
        auto_ptr<CC> m_c;
    };
    // This class will be the argument to the iterator. It contains 4
    // instances of CA - directly, through pointers, and via inheritance
    class CX
    {
    public:
        CX() : m_a(0), m_b(0), m_d(0) {};
        ~CX(){};
        static const CTypeInfo* GetTypeInfo(void);
        auto_ptr<CA> m_a; // auto_ptr to a CA
        CB *m_b;          // pointer to an object containing a CA
        CC m_c;           // instance of a subclass of CA
        CRef<CD> m_d;      // CRef to an object containing an auto_ptr to CC
    };
    // Implement the GetTypeInfo() methods
    // (see User-defined type information)
    BEGIN_CLASS_INFO(CA)
    {
        ADD_STD_MEMBER(m_Data);
        ADD_SUB_CLASS(CC);
    }
    END_CLASS_INFO

```

```

BEGIN_CLASS_INFO(CB)
{
    ADD_MEMBER(m_a, STL_auto_ptr, (CLASS, (CA)));
}
END_CLASS_INFO

BEGIN_DERIVED_CLASS_INFO(CC, CA)
{
}
END_DERIVED_CLASS_INFO

BEGIN_CLASS_INFO(CD)
{
    ADD_MEMBER(m_c, STL_auto_ptr, (CLASS, (CC)));
}
END_CLASS_INFO

BEGIN_CLASS_INFO(CX)
{
    ADD_MEMBER(m_a, STL_auto_ptr, (CLASS, (CA)));
    ADD_MEMBER(m_b, POINTER, (CLASS, (CB)));
    ADD_MEMBER(m_c, CLASS, (CC));
    ADD_MEMBER(m_d, STL_CRef, (CLASS, (CD)));
}
END_CLASS_INFO

int main(int argc, char** argv)
{
    CB b;
    CD d;

    b.m_a.reset(new CA(2));
    d.m_c.reset(new CC(4));
    CX x;

    x.m_a.reset(new CA(1));    // auto_ptr to CA
    x.m_b = &b;                // pointer to CB containing auto_ptr to CA
    x.m_c = *(new CC(3));      // instance of subclass of CA
    x.m_d = &d;                // CRef to CD containing auto_ptr to CC

    cout << "Iterating over CA objects in x" << endl << endl;

    for (CTypeIterator<CA> i(Begin(x)); i; ++i)
        cout << (*i).m_Data << endl;

    cout << "Iterating over CC objects in x" << endl << endl;

    for (CTypeIterator<CC> i(Begin(x)); i; ++i)
        cout << (*i).m_Data << endl;

```

```

    cout << "Iterating over CObjects in x" << endl << endl;
    for (CObjectIterator i(Begin(x)); i; ++i) {
        const CD *tmp = dynamic_cast<const CD*>(&*i);
        cout << tmp->m_c->m_Data << endl;
    }
    return 0;
}

```

Figure 2 illustrates the paths traversed by **CTypeIterator<CA>** and **CTypeIterator<CC>**, where both iterators are initialized with **Begin(a)**. The data members visited by the iterator are indicated by enclosing boxes. See Figure 2.

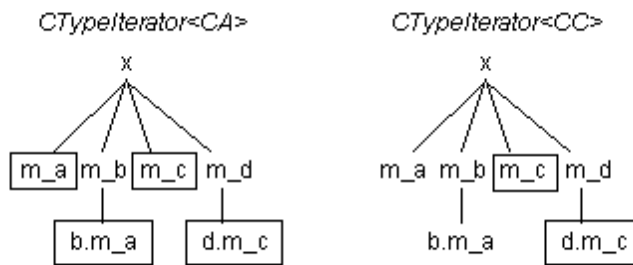


Figure 2

**Figure 2:** Traversal path of the CTypeIterator

For additional examples of using the type iterators described in this section, see *ctypeiter.cpp*.

## Processing Serial Data

Although this discussion focuses on ASN.1 and XML formatted data, the data structures and tools described here have been designed to (potentially) support any formalized serial data specification. Many of the tools and objects have open-ended abstract or template implementations that can be instantiated differently to fit various specifications.

The following topics are discussed in this section

- Accessing the object header files and serialization libraries
- Reading and writing serial data
- Determining Which Header Files to Include
- Determining Which Libraries to Link To

## Accessing the object header files and serialization libraries

Reading and writing serialized data is implemented by an integrated set of streams, filters, and object types. An application that reads encoded data files will require the object header files and libraries which define how these serial streams of data should be loaded into memory. This entails `#include` statements in your source files, as well as the associated library specifications in your makefiles. The object header and implementation files are located in the `include/objects` and `src/objects` subtrees of the C++ tree, respectively. The header and implementation files for serialized streams and type information are in the `include/serial` and `src/serial` directories.

If you have checked out the `objects` directories, but not explicitly run the `datatool` code generator, then you will find that your `include/objects` subdirectories are (almost) empty, and the source subdirectories contain only makefiles and ASN.1 specifications. These makefiles and ASN.1 specifications can be used to build your own copies of the objects' header and implementation files, using `make all_r` (if you configured using the `--with-objects` flag), or running `datatool` explicitly.

However, building your own local copies of these header and implementation files is neither necessary nor recommended, as it is simpler to use the pre-generated header files and prebuilt libraries. The pre-built header and implementation files can be found in `$NCBI/c++/include/objects/` and `$NCBI/c++/src/objects/`, respectively. Assuming your makefile defines an include path to `$NCBI/c++/include`, selected object header files such as `Date.hpp`, can be included as:

```
#include <OBJECTS Date.hpp general>
```

This header file (along with its implementations in the accompanying `src` directory) was generated by `datatool` using the specifications from `src/objects/general/general.asn`. In order to use the classes defined in the `objects` directories, your source code should begin with the statements:

```
USING_NCBI_SCOPE;
using namespace objects;
```

All of the objects' header and implementation files are generated by `datatool`, as specified in the ASN.1 specification files. The resulting object definitions however, are not in any way dependent on ASN.1 format, as they simply specify the in-memory representation of the defined data types. Accordingly, the objects themselves can be used to read, interpret, and write any type of serialized data. Format specializations on the input stream are implemented via ***CObjectInputStream*** objects, which extract the required tags and values from the input data according to the format specified. Similarly, Format specializations on an output stream are implemented via ***CObjectOutputStream*** objects.

## Reading and writing serial data

Let's consider a program `xml2asn.cpp` that translates an XML data file containing an object of type `Biostruc`, to ASN.1 text and binary formats. In `main()`, we begin by initializing the diagnostic stream to write errors to a local file called `xml2asn.log`. (Exception handling, program tracing, and error logging are described in the Diagnostic Streams section).

An instance of the **CTestAsn** class is then created, and its member function **AppMain()** is invoked. This function in turn calls **CTestAsn::Run()**. The first three lines of code there define the XML input and ASN.1 output streams, using *auto\_ptrs*, to ensure automatic destruction of these objects.

Each stream is associated with data serialization mechanisms appropriate to the **ESerial-DataFormat** provided to the constructor:

```
enum ESerialDataFormat {
    eSerial_None      = 0,
    eSerial_AsnText    = 1,    // open ASN.1 text format
    eSerial_AsnBinary  = 2,    // open ASN.1 binary format
    eSerial_Xml        = 3     // open XML format (not supported yet)
};
```

**CObjectIStream** and **CObjectOStream** are base classes which provide generic interfaces between the specific type information of a serializable object and an I/O stream. The object stream classes that will actually be instantiated by this application, **CObjectIStreamXml**, **CObjectOStreamAsn**, and **CObjectOStreamAsnBinary**, are descendants of these base classes.

Finally, a variable for the object type that will be generated from the input stream (in this case a **CBiostruc**) is defined, and the **CObject[I/O]Stream** operators "<<" and ">>" are used to read and write the serialized data to and from the object. (Note that it is **not** possible to simply "pass the data through", from the input stream to the output stream, using a construct like: *\*inObject >> \*outObject*). The **CObject[I/O]Streams** know nothing about the structure of the specific object - they have knowledge only of the serialization format (text ASN, binary ASN, XML, etc.). In contrast, the **CBiostruc** knows nothing about I/O and serialization formats, but it contains explicit type information about itself. Thus, the **CObject[I/O]Streams** can apply their specialized serialization methods to the data members of **CBiostruc** using the *type information* associated with that object's class.

## Determining Which Header Files to Include

As always, we include the `corelib` header files, *ncbistd.hpp* and *ncbiapp.hpp*. In addition, the *serial* header files that define the generic **CObject[I/O]Stream** objects are included, along with *serial.hpp*, which defines generalized serialization mechanisms. Finally, we need to include the header file for the object type we will be using.

There are two source browsers that can be used to locate the appropriate header file for a particular object type. All class names in the NCBI C++ Toolkit begin with the letter "C". Using the class hierarchy browser, we find **CBiostruc**, derived from **CBiostruc\_Base**, which is in turn derived from **CObject**. Following the [CBiostruc](#) link, we can then use the *locate* button to move to the LXR source code navigator, and there, find the name of the header file. In this case, we find *CBiostruc.hpp* is located in *include/objects/mmdb1*. Alternatively, if we know the name of the C++ class, the source code navigator's identifier search tool can be used directly. In summary, the following `#include` statements appear at the top of *xml2asn.cpp*:

```
#include <CORELIB ncbistd.hpp>
#include <CORELIB ncbiapp.hpp>
#include <SERIAL serial.hpp>
```

```
#include <SERIAL objistr.hpp>
#include <SERIAL objostr.hpp>
#include <OBJECTS Biostruc.hpp mmdb1>
```

## Determining Which Libraries to Link To

Determining which libraries must be linked to requires a bit more work and may involve some trial and error. The list of available libraries currently includes:

access biblio cdd featdef general medlars medline mmdb1 mmdb2 mmdb3 ncbimime objprt  
proj pub pubmed seq seqalign seqblock seqcode seqfeat seqloc seqres seqset submit xcgi xcon-  
nect xfcgi xhtml xncbi xser

It should be clear that we will need to link to the core library, `xncbi`, as well as to the serial library, `xser`. In addition, we will need to link to whatever object libraries are entailed by using a **CBiostruc** object. Minimally, one would expect to link to the `mmdb` libraries. This in itself is insufficient however, as the **CBiostruc** class embeds other types of objects, including PubMed citations, features, and sequences, which in turn embed additional objects such as **Date**. The makefile for `xml2asn.cpp`, `Makefile.xml2asn.app` lists the libraries required for linking in the make variable `LIB`.

```
#####
###
# This file was originally generated from by shell script "new_project.sh"
#####
APP = xml2asn
OBJ = xml2asn
LIB = mmdb1 mmdb2 mmdb3 seqloc seqfeat pub medline biblio general xser
xncbi
LIBS = $(NCBI_C_LIBPATH) -lncbi $(ORIG_LIBS)
```

See also the example program, `asn2asn.cpp` which demonstrates more generalized translation of **Seq-entry** and **Bioseq-set** (defined in `seqset.asn`).

## User-defined type information

---

The following topics are discussed in this section:

- Introduction
- Installing a `GetTypeInfo()` function: the `BEGIN_/END_` macros
- Specifying internal structure and class inheritance: the `ADD_` macros

## Introduction

Object type information, as it is used in the NCBI C++ Toolkit, is defined in the section on Runtime Object Type Information. As described there, all of the classes and constructs defined in the serial *include* and *src* directories have a static implementation of a **GetTypeInfo()** function that yields a **CTypeInfo** for the object of interest. In this section, we describe how type information can also be generated and accessed for user-defined types. We begin with a review of some of the basic notions introduced in the previous discussion.

The type information for a class is stored outside any instances of that class, in a statically created **CTypeInfo** object. A class's type information includes the class layout, inheritance relations, external alias, and various other attributes that are independent of specific instances. In addition, the type information object provides an interface to the class's data members.

Limited type information is also available for primitive data types, enumerations, containers, and pointers. The type information for a primitive type specifies that it is an **int**, **float**, or **char**, etc., and whether or not that element is signed. Enumerations are a special kind of primitive type, whose type information specifies its enumeration values and named elements. Type information for containers can specify both the type of container and the type of elements. The type information for a pointer provides convenient methods of access to the type information for the type pointed to.

For all types, the type information is encoded in a static **CTypeInfo** object, which is then accessed by all instances of a given type using a **GetTypeInfo()** function. For class types, this function is implemented as a static method for the class. For non class types, **GetTypeInfoXxx()** is implemented as a static global function, where *Xxx* is a unique suffix generated from the type's name. With the first invocation of **GetTypeInfo()** for a given type, the static **CTypeInfo** object is created, which then persists (local to the function **GetTypeInfo()**) throughout execution. Subsequent calls to **GetTypeInfo()** simply return a pointer to this statically created local object.

In order to make type information about *user-defined* classes accessible to your application, the user-defined classes must also implement a static **GetTypeInfo()** method. A set of preprocessor macros is available, which greatly simplifies this effort. A pre-requisite to using these macros however, is that the class definition must include the following line:

```
DECLARE_INTERNAL_TYPE_INFO();
```

This pre-processor macro will generate the following in-line statement in the class definition:

```
static const NCBI_NS_NCBI::CTypeInfo* GetTypeInfo(void);
```

As with class objects, there must be some means of declaring the type information function for an enumeration prior to using the macros which implement that function. Given an enumeration named **EMyEnum**, **DECLARE\_ENUM\_INFO(EMyEnum)** will generate the following declaration:

```
const CEnumeratedTypeValues* GetTypeInfo_enum_EMyEnum(void);
```

The `DECLARE_ENUM_INFO()` macro should appear in the header file where the enumeration is defined, immediately following the definition. The `DECLARE_INTERNAL_ENUM_INFO` macro is intended for usage with internal class definitions, as in:

```
class ClassWithEnum {
    enum EMyEnum {
        ...
    };

    DECLARE_INTERNAL_ENUM_INFO(EMyEnum);
    ...
};
```

The C++ Toolkit also allows one to provide type information for legacy C style *struct* and *choice* elements defined in the C Toolkit. The mechanisms used to implement this are mentioned but not described in detail here, as it is not likely that newly-defined types will be in these categories.

## Installing a `GetTypeInfo()` function: the `BEGIN_/END_` macros

Several pre-processor macros are available for the installation of the ***GetTypeInfo()*** functions for different types. Table 2 lists six `BEGIN_NAMED_*_INFO` macros, along with a description of the type of object each can be applied to and its expected arguments. Each macro in Table 2 has a corresponding `END_*_INFO` macro definition.

The first four macros in Table 2 apply to C++ objects. The `DECLARE_INTERNAL_TYPE_INFO()` macro **must** appear in the class definition's public section. These macros take two **string** arguments:

- an external alias for the type, and
- the internal C++ symbolic class name.

The external alias is required for serializable objects whose external name differs from the internal C++ class name. For example, the external object names specified in the ASN.1 modules (in *src/objects*) are prefixed with the letter "C" in the corresponding C++ class names (e.g., *Bioseq* versus ***CBioseq***). Each of the "named" macros in Table 2 has a corresponding "unnamed" macro which accepts the (unquoted) symbolic class name as one of its arguments, and generates a call to the corresponding "named" macro using a quoted string. For example, `BEGIN_CLASS_INFO` is defined as:

```
#define BEGIN_CLASS_INFO(className) \
    BEGIN_NAMED_CLASS_INFO(#className, className)
```

The next two macros implement global, uniquely named functions which provide access to type information for C++ enumerations; the resulting functions are named *GetTypeInfo\_enum\_[EnumName]*. The `DECLARE_ENUM_INFO( )` or `DECLARE_ENUM_INFO_IN( )` macro should be used in these cases to declare the **GetTypeInfo\*()** functions.

The usage of these six macros generally takes the following form:

```
BEGIN_*_INFO(ClassName)
{
    ADD_*(MemberName);
    ADD_*(memberName);
    ...
}
END_*_INFO
```

That is, the `BEGIN/END` macros are used to generate the function's signature and enclosing block, and various `ADD_*` macros are applied to add information about internal members and class relations.

**Table 2. BEGIN\_NAMED\_\* Macro names and their usage**

Macro name	Used for	Arguments
<code>BEGIN_NAMED_CLASS_INFO</code>	Non-abstract class object	<code>ClassAlias</code> , <code>ClassName</code>
<code>BEGIN_NAMED_ABSTRACT_CLASS_INFO</code>	Abstract class object	<code>ClassAlias</code> , <code>ClassName</code>
<code>BEGIN_NAMED_DERIVED_CLASS_INFO</code>	Derived subclass object	<code>ClassAlias</code> , <code>ClassName</code> , <code>BaseClassName</code>
<code>BEGIN_NAMED_CHOICE_INFO</code>	C++ class choice object	<code>ClassAlias</code> , <code>ClassName</code>
<code>BEGIN_NAMED_ENUM_INFO</code>	Enum object	<code>EnumAlias</code> , <code>EnumName</code> , <code>IsInteger</code>
<code>BEGIN_NAMED_ENUM_IN_INFO</code>	internal Enum object	<code>EnumAlias</code> , <code>CppContext</code> , <code>EnumName</code> , <code>IsInteger</code>

- `BEGIN_NAMED_CLASS_INFO(ClassAlias, ClassName)BEGIN_CLASS_INFO(ClassName)` These macros should be used on classes that do not contain any pure virtual functions. For example, the **GetTypeInfo()** method for the **CPerson** class (used in the chapter on iterators) can be implemented as:

```
BEGIN_NAMED_CLASS_INFO("CPerson", CPerson)
{
    ADD_NAMED_STD_MEMBER("m_Name", m_Name);
    ADD_NAMED_STD_MEMBER("m_Addr", m_Addr);
    ADD_NAMED_MEMBER("m_NextDoor", m_NextDoor, POINTER, (CLASS, (CPerson)));
}
END_CLASS_INFO
```

or, equivalently, as:

```

BEGIN_CLASS_INFO(CPerson)
{
    ADD_STD_MEMBER(m_Name);
    ADD_STD_MEMBER(m_Addr);
    ADD_MEMBER(m_NextDoor, POINTER, (CLASS, (CPerson)));
}
END_CLASS_INFO

```

Here, the **CPerson** class has two **string** data members, `m_Name` and `m_Addr`, as well as a pointer to an object of the same type (**CPerson\***). All built-in C++ types such as **int**, **float**, **string** etc., use the `ADD_NAMED_STD_MEMBER` or `ADD_STD_MEMBER` macros. These and other macros used to add members are defined in Specifying internal structure and class inheritance: the `ADD_` macros and Table 3.

- `BEGIN_NAMED_ABSTRACT_CLASS_INFO(ClassAlias, ClassName)`  
`BEGIN_ABSTRACT_CLASS_INFO(ClassName)` These macros must be used on abstract base classes which contain pure virtual functions. Because these abstract classes cannot be instantiated, special handling is required in order to install their static **GetTypeInfo()** methods.
- `BEGIN_NAMED_DERIVED_CLASS_INFO(ClassAlias, ClassName, BaseClassName)`  
`BEGIN_DERIVED_CLASS_INFO(ClassName, BaseClassName)` These macros should be used on derived subclasses whose parent classes also have the **GetTypeInfo()** method implemented. Data members inherited from parent classes should not be included in the derived class type information.

```

BEGIN_DERIVED_CLASS_INFO(CA, CBase)
{
    // ... data members in CA not inherited from CBase
}
END_DERIVED_CLASS_INFO
BEGIN_DERIVED_CLASS_INFO(CB, CBase)
{
    // ... data members in CB not inherited from CBase
}
END_DERIVED_CLASS_INFO

```

**NOTE:** The type information for classes derived directly from **CObject** does **not** however, follow this protocol. In this special case, although the class is derived from **CObject**, you should **not** use the `DERIVED_CLASS` macros to implement **GetTypeInfo()**, but instead use the usual `BEGIN_CLASS_INFO` macro. **CObject**'s have a slightly different interface to their type information (see **CObjectGetTypeInfo**), and apply these macros differently.

- `BEGIN_NAMED_CHOICE_INFO(ClassAlias, ClassName)`  
`BEGIN_CHOICE_INFO(ClassName)` These macros install **GetTypeInfo()** for C++ *choice* objects, which are implemented as C++ classes. See Choice objects in the C++ Toolkit for a description of C++ *choice* objects. Each of the choice variants occurs as a data member in the class, and

the macros used to add choice variants (`ADD_NAMED_%20_CHOICE_VARIANT`) are used similarly to those which add data members to classes (see discussion of the `ADD*` macros below).

- `BEGIN_NAMED_ENUM_INFO(EnumAlias, EnumName, IsInteger)`  
`BEGIN_ENUM_INFO(EnumName, IsInteger)` In addition to the two arguments used by the `BEGIN_*_INFO` macros for classes, a Boolean argument (***IsInteger***) indicates whether or not the enumeration includes arbitrary integer values or only those explicitly specified.
- `BEGIN_NAMED_ENUM_IN_INFO(EnumAlias, CppContext, EnumName, IsInteger)`  
`BEGIN_ENUM_IN_INFO(CppContext, EnumName, IsInteger)` These macros also implement the type information functions for C++ enumerations --but in this case, the enumeration is defined outside the scope where the macro is applied, so a *context* argument is required. This new argument, ***CppContext***, specifies the C++ class name or external namespace where the enumeration is defined.

Again, when using the above macros to install type information, the corresponding class definitions **must** include a declaration of the static class member function ***GetTypeInfo()*** in the class's public section. The `DECLARE_INTERNAL_TYPE_INFO( )` macro is provided to ensure that the declaration of this method is correct. Similarly, the `DECLARE_INTERNAL_ENUM_INFO` and `DECLARE_ENUM_INFO` macros should be used in the header files where enumerations are defined. The `DECLARE_ASN_TYPE_INFO` and `DECLARE_ASN_CHOICE_INFO` macros can be used to declare the type information functions for C-style structs and choice nodes.

## Specifying internal structure and class inheritance: the `ADD_*` macros

Information about internal class structure and inheritance is specified using the `ADD_*` macros (see Table 3). Again, each macro has both a "named" and "unnamed" implementation. The arguments to all of the `ADD_NAMED_*` macros begin with the external alias and C++ name of the item to be added.

The `ADD_*` macros that take **only** an alias and a name require that the type being added must be either a built-in type or a type defined by the name argument. When adding a ***CRef*** data member to a class or choice object however, the class referenced by the ***CRef*** must be made explicit with the `RefClass` argument, which is the C++ class name for the type pointed to.

Similarly, when adding an enumerated data member to a class, the enumeration itself must be explicitly named. For example, if class ***CMyClass*** contains a data member `m_MyEnumVal` of type ***EMyEnum***, then the `BEGIN_NAMED_CLASS_INFO` macro for ***CMyClass*** should contain the statement:

```
ADD_ENUM_MEMBER (m_MyEnumVal, EMyEnum);
```

or, equivalently:

```
ADD_NAMED_ENUM_MEMBER ("m_MyEnumVal", m_MyEnumVal, EMyEnum);
```

or, to define a "custom" (non-default) external alias:

```
ADD_NAMED_ENUM_MEMBER ("m_CustomAlias", m_MyEnumVal, EMyEnum);
```

Here, **EMyEnum** is defined in the same namespace and scope as **CMyClass**. Alternatively, if the enumeration is defined in a different class or namespace (and therefore, then the `ADD_ENUM_IN_MEMBER` macro must be used:

```
ADD_ENUM_IN_MEMBER (m_MyEnumVal, COtherClassName::, EMyEnum);
```

In this example, **EMyEnum** is defined in a class named **COtherClassName**. The **CppContext** argument (defined here as **COtherClassName::**) acts as a scope operator, and can also be used to specify an alternative namespace. The `ADD_NAMED_ENUM_CHOICE_VARIANT` and `ADD_NAMED_ENUM_IN_CHOICE_VARIANT` macros are used similarly to provide information about enumerated choice options. The `ADD_ENUM_VALUE` macro is used to add enumerated values to the enumeration itself, as demonstrated in the above example of the `BEGIN_NAMED_ENUM_INFO` macro.

**Table 3. ADD\_\* Macros and their usage**

Macro name	Usage	Arguments
<code>ADD_NAMED_STD_MEMBER</code>	Add a standard data member to a class	MemberAlias, Member-Name
<code>ADD_NAMED_CLASS_MEMBER</code>	Add an internal class member to a class	MemberAlias, Member-Name
<code>ADD_NAMED_SUB_CLASS</code>	Add a derived subclass to a class	SubClassAlias, Sub-ClassName
<code>ADD_NAMED_REF_MEMBER</code>	Add a <b>CRef</b> data member to a class	MemberAlias, Member-Name, RefClass
<code>ADD_NAMED_ENUM_MEMBER</code>	Add an enumerated data member to a class	MemberAlias, Member-Name, EnumName
<code>ADD_NAMED_ENUM_IN_MEMBER</code>	Add an externally defined enumerated data member to a class	MemberAlias, Member-Name, CppContext, EnumName
<code>ADD_NAMED_MEMBER</code>	Add a data member of the type specified by <code>TypeMacro</code> to a class	MemberAlias, Member-Name, <code>TypeMacro</code> , <code>TypeMacroArgs</code>
<code>ADD_NAMED_STD_CHOICE_VARIANT</code>	Add a standard variant type to a C++ choice object	VariantAlias, VariantName
<code>ADD_NAMED_REF_CHOICE_VARIANT</code>	Add a <b>CRef</b> variant to a C++ choice object	VariantAlias, Variant-Name, RefClass
<code>ADD_NAMED_ENUM_CHOICE_VARIANT</code>	Add an enumeration variant to a C++ choice object	VariantAlias, Variant-Name, EnumName

Macro name	Usage	Arguments
ADD_NAMED_ENUM_IN_CHOICE_VARIANT	Add an enumeration variant to a C++ choice object	VariantAlias, Variant-Name, CppContext, EnumName
ADD_NAMED_CHOICE_VARIANT	Add a variant of the type specified by TypeMacro to a C++ choice object	VariantAlias, Variant-Name, TypeMacro, TypeMacroArgs
ADD_ENUM_VALUE	Add a named enumeration value to an <i>enum</i>	EnumValName, Value

The most complex macros by far are those which use the **TypeMacro** and TypeMacroArgs arguments: ADD(\_NAMED)\_MEMBER and ADD(\_NAMED)\_CHOICE\_VARIANT. These macros are more open-ended and allow for more complex specifications. We have already seen one example of using a macro of this type, in the implementation of the **GetTypeInfo()** method for **CPerson**:

```
ADD_MEMBER(m_NextDoor, POINTER, (CLASS, (CPerson)));
```

The ADD\_MEMBER and ADD\_CHOICE\_VARIANT macros always take at least two arguments:

1. the internal member (variant) name
2. the definition of the member's (variant's) type

Depending on the (second) TypeMacro argument, additional arguments may or may not be needed. In this example, the TypeMacro is **POINTER**, which **does require** additional arguments. The TypeMacroArgs here specify that m\_NextDoor is a pointer to a class type whose C++ name is **CPerson**.

More generally, the remaining arguments depend on the value of TypeMacro, as these parameters complete the type definition. The possible strings which can occur as TypeMacro, along with the additional arguments required for that type, are given in Table 4.

**Table 4. Type macros and their arguments**

TypeMacro	TypeMacroArgs
CLASS	(ClassName)
STD	(C++ type)
StringStore	()
null	()
ENUM	(EnumType, EnumName)
POINTER	(Type,Args)
STL_multiset	(Type,Args)
STL_set	(Type,Args)

TypeMacro	TypeMacroArgs
STL_multimap	(KeyType,KeyArgs,ValueType,ValueArgs)
STL_map	(KeyType,KeyArgs,ValueType,ValueArgs)
STL_list	(Type,Args)
STL_list_set	(Type,Args)
STL_vector	(Type,Args)
STL_CHAR_vector	(C++ Char type)
STL_auto_ptr	(Type,Args)
CHOICE	(Type,Args)

The `ADD_MEMBER` macro generates a call to the corresponding `ADD_NAMED_MEMBER` macro as follows:

```
#define ADD_MEMBER(MemberName,TypeMacro,TypeMacroArgs) \
    ADD_NAMED_MEMBER(#MemberName,MemberName,TypeMacro,TypeMacroArgs)
```

Some examples of using the `ADD_MEMBER` macro are:

```
ADD_MEMBER(m_X) ;
ADD_MEMBER(m_A, STL_auto_ptr, (CLASS, (ClassName)));
ADD_MEMBER(m_B, STL_CHAR_vector, (char));
ADD_MEMBER(m_C, STL_vector, (STD, (int)));
ADD_MEMBER(m_D, STL_list, (CLASS, (ClassName)));
ADD_MEMBER(m_E, STL_list, (POINTER, (CLASS, (ClassName))));
ADD_MEMBER(m_F, STL_map, (STD, (long), STD, (string)));
```

Similarly, the `ADD_CHOICE_VARIANT` macro generates a call to the corresponding `ADD_NAMED_CHOICE_VARIANT` macro. These macros add type information for the choice object's variants.

## Runtime Object Type Information

The following topics are discussed in this section:

- Introduction
- Motivation
- Object Information Classes
- Usage of object type information

### Introduction

Run-time information about data types is necessary in several contexts, including:

1. When reading, writing, and processing serialized data, where runtime information about a type's internal structure is needed

2. When reading from an arbitrary data source, where data members' external aliases must be used to locate the corresponding class data members (e.g. *MyXxx* may be aliased as *my-xxx* in the input data file)
3. When using a generalized C++ type iterator to traverse the data members of an object
4. When accessing the object type information per se (without regard to any particular object instance), e.g. to dump it to a file as ASN.1 or DTD specifications (not *data*)

In the first three cases above, it is necessary to have both the object itself as well as its runtime type information. This is because in these contexts, the object is usually passed inside a generic function, as a pointer to its most base parent type *CObject*. The runtime type information is needed here, as there is no other way to ascertain the actual object's data members. In addition to providing this information, a runtime type information object provides an interface for accessing and modifying these data members.

In the last case (4) above, the type information is used independent of any actual object instances.

The NCBI C++ Toolkit uses two classes to support these requirements:

- **Type information classes** (base class **>CTypeInfo**) are intended for internal usage only, and they encode information about a type, devoid of any instances of that type. This information includes the class layout, inheritance relations, external alias, and various other attributes such as size, which are independent of specific instances. Each data member of a class also has its own type information. Thus, in addition to providing information relevant to the member's occurrence in the class (e.g. the member name and offset), the type information for a *class* must also provide access to the type information for each of its *members*. Limited type information is also available for types other than classes, such as primitive data types, enumerations, containers, and pointers. For example, the type information for a primitive type specifies that it is an *int*, *float*, or *char*, etc., and whether or not that element is signed. Enumerations are a special kind of primitive type, whose type information specifies its enumeration values and named elements. Type information for containers specifies both the type of container and the type of elements that it holds.
- **Object information classes** (base class **CObjectTypeInfo**) include a pointer to the type information as well as a pointer to the object *instance*, and provide a safe interface to that object. In situations where type information is used independent of any concrete object, the object information class simply serves as a wrapper to a type information object. Where access to an object instance is required, the object pointer provides direct access to the correctly type-cast instance, and the interface provides methods to access and/or modify the object itself or members of that object.

The C++ Toolkit stores the type information outside any instances of that type, in a statically created **CTypeInfo** object. For class objects, this **CTypeInfo** object can be accessed by all instances of the class via a static **GetTypeInfo()** class method. Similarly, for primitive types and other constructs that have no way of associating methods with them per se, a static globally defined **GetTypeInfoXxx()** function is used to access a static **CTypeInfo** object. (The Xxx suffix is used here to indicate that a globally unique name is generated for the function).

All of the automatically generated classes and constructs defined in the C++ Toolkit's *objects/* directory already have static **GetTypeInfo()** functions implemented for them. In order to make type information about *user-defined* classes and elements also accessible, you will need to implement static **GetTypeInfo()** functions for these constructs. A number of pre-processor macros are available to support this activity, and are described in the section on User-defined Type Information.

Type information is often needed when the object itself has been passed anonymously, or as a pointer to its parent class. In this case, it is not possible to invoke the **GetTypeInfo()** method directly, as the object's exact type is unknown. Using a `<static_cast>` operator to enable the member function is also unsafe, as it may open the door to incorrectly associating an object's pointer with the wrong type information. For these reasons, the **CTypeInfo** class is intended for internal usage only, and it is the **CObjectTypeInfo** classes that provide a more safe and friendly user interface to type information.

## Motivation

We use a simple example to help motivate the use of this *type* and *object* information model. Let us suppose that we would like to have a generic function **LoadObject()**, which can populate an object using data read from a flat file. For example, we might like to have:

```
bool LoadObject(Object& myObj, istream& is);
```

where `myObj` is an instance of some subclass of **Object**. Assuming that the text in the file is of the form:

```
MemberName1 value1
MemberName5 value5
MemberName2 value2
:
```

we would like to find the corresponding data member in `myObj` for each `MemberName`, and set that data member's value accordingly. Unfortunately, `myObj` cannot directly supply any useful type information, as the member names we seek are for a specific subclass of **Object**. Now suppose that we have an appropriate type information object available for `myObj`, and consider how this might be used:

```
bool LoadObject(TypeInfo& info, Object& myObj, istream& is)
{
    string myName, myValue;

    while ( !is.eof() ) {
        is >> myName >> myValue;
```

```

        void* member = FindMember(info, myObj, myName);
        AssignValue(member, myValue);
    }
}

```

Here, we assume that our type information object, `info`, stores information about the memory offset of each data member in `myObj`, and that such information can be retrieved using some sort of identifying member name such as `myName`. This is not too difficult to imagine, and indeed, this is exactly the type of information and facility provided by the C++ Toolkit's type information classes. The ***FindMember()*** function just needs to return a ***void*** pointer to the appropriate location in memory. The ***AssignValue()*** function presents a much greater challenge however, as its two sole arguments are a ***void*** pointer and a ***string***. This would be fine if the data member was indeed a ***void*** pointer, and a ***string*** value was acceptable. In general this is not the case, and stronger methods are clearly needed.

In particular, for each data member encountered, we need to retrieve the type of that member as well as its location in memory, so as to process `myValue` appropriately before assigning it. In addition, we need safer mechanisms for making such "untyped" assignments. Ideally, we would like a ***FindMember()*** function that returns a correctly cast pointer to that data member, along with its associated type information. This is what the object information classes provide - a pointer to the object instance as well as a pointer to its static *type* information. The interface to the ***object*** information class also provides a number of methods such as ***GetClassMember()***, ***GetTypeFamily()***, ***SetPrimitiveValue()***, etc., to support the type of activity described above.

## Object Information Classes

The following topics are discussed in this section:

- `CObjectTypeInfo (*)`
- `CConstObjectInfo (*)`
- `CObjectInfo (*)`

### *CObjectTypeInfo* (%20)

This is the base class for all *object* information classes. It is intended for usage where there is no concrete object being referenced, and all that is required is access to the type information. A ***CObjectTypeInfo*** contains a pointer to a low-level ***CTypeInfo*** object, and functions as a user-friendly wrapper class.

The constructor for ***CObjectTypeInfo*** takes a pointer to a *const* ***CTypeInfo*** object as its single argument. This is precisely what is returned by all of the static ***GetTypeInfo()*** functions. Thus, to create a ***CObjectTypeInfo*** for the ***CBioseq*** class - without reference to any particular instance of ***CBioseq*** - one might use:

```
CObjectTypeInfo objInfo( CBioseq::GetTypeInfo() );
```

One of the most important methods provided by the **CObjectTypeInfo** class interface is **GetTypeFamily()**, which returns an enumerated value indicating the *type family* for the object of interest. Five type families are defined by the **ETypeFamily** enumeration:

```
ETypeFamily GetTypeFamily(void) const;
enum ETypeFamily {
    eTypeFamilyPrimitive,
    eTypeFamilyClass,
    eTypeFamilyChoice,
    eTypeFamilyContainer,
    eTypeFamilyPointer
};
```

Different queries become appropriate depending on the **ETypeFamily** of the object. For example, if the object is a container, one might need to determine the type of container (e.g. whether it is a *list*, *map* etc.), and the type of element. Similarly, if an object is a primitive type (e.g. *int*, *float*, *string*, etc.), an appropriate query becomes what the value type is, and in the case of integer-valued types, whether or not it is signed. Finally, in the case of more complex objects such as class and choice objects, access to the type information for the individual data members and choice variants is needed. The following methods are included in the **CObjectTypeInfo** interface for these purposes:

- *GetTypeFamily() == eTypeFamilyPrimitive:*
  - *EPrimitiveValueType GetPrimitiveValueType(void) const;*
  - *bool IsPrimitiveValueSigned(void) const;*
- *GetTypeFamily() == eTypeFamilyClass:*
  - *CMemberIterator BeginMembers(void) const;*
  - *CMemberIterator FindMember(const string& memberName) const;*
  - *CMemberIterator FindMemberByTag(int memberTag) const;*
- *GetTypeFamily() == eTypeFamilyChoice:*
  - *CVariantIterator BeginVariants(void) const;*
  - *CVariantIterator FindVariant(const string& memberName) const;*
  - *CVariantIterator FindVariantByTag(int memberTag) const;*
- *GetTypeFamily() == eTypeFamilyContainer:*
  - *EContainerType GetContainerType(void) const;*

- *CObjectTypeInfo GetElementType(void) const;*
- *GetTypeFamily() == eTypeFamilyPointer.*
- *CObjectTypeInfo GetPointedType(void) const;*

The two additional enumerations referred to here, **EContainerType** and **EPrimitiveValueType**, are defined, along with **ETypeFamily**, in *include/serial/serialdef.hpp*.

Different iterator classes are used for iterating over class data members versus choice variant types. Thus, if the object of interest is a C++ class object, then access to the type information for its members can be gained using a **CObjectTypeInfo::CMemberIterator**. The **BeginMembers()** method returns a **CMemberIterator** pointing to the first data member in the class; the **FindMember\*()** methods return a **CMemberIterator** pointing to a data member whose name or tag matches the input argument. The **CMemberIterator** class is a forward iterator whose operators are defined as follows:

- the ++ operator increments the iterator (makes it point to the next class member)
- the () operator tests that the iterator has not exceeded the legitimate range
- the \* dereferencing operator returns a **CObjectTypeInfo** for the data member the iterator currently points to

Similarly, the **BeginVariants()** and **FindVariant()** methods allow iteration over the choice variant data types for a choice class, and the dereferencing operation yields a **CObjectTypeInfo** object for the choice variant currently pointed to by the iterator.

## CConstObjectInfo (%20)

The **CConstObjectInfo** (derived from **CObjectTypeInfo**) adds an interface to access the particular instance of an object (in addition to the interface inherited from **CObjectTypeInfo**, which provides access to type information only). It is intended for usage with *const* instances of the object of interest, and therefore the interface does not permit any modifications to the object. The constructor for **CConstObjectInfo** takes two arguments:

```
CConstObjectInfo(const void* instancePtr, const CTypeInfo* typeinfoPtr);
```

(Alternatively, the constructor can be invoked with a single STL pair containing these two objects.)

Each **CConstObjectInfo** contains a pointer to the object's type information as well as a pointer to an instance of the object. The existence or validity of this instance can be checked using any of the following **CConstObjectInfo** methods and operators:

- *bool Valid(void) const;*

- *operator bool(void) const;*
- *bool operator!(void) const;*

For *primitive* type objects, the **CConstObjectInfo** interface provides access to the currently assigned value using **GetPrimitiveValueXxx()**. Here, *Xxx* may be *Bool*, *Char*, *Long*, *ULong*, *Double*, *String*, *ValueString*, or *OctetString*. In general, to get a primitive value, one first applies a *switch* statement to the value returned by **GetPrimitiveValueType()**, and then calls the appropriate **GetPrimitiveValueXxx()** method depending on the branch followed, e.g.:

```
switch ( obj.GetPrimitiveValueType() ) {
case ePrimitiveValueBool:
    bool b = obj.GetPrimitiveValueBool();
    break;

case ePrimitiveValueInteger:
    if ( obj.IsPrimitiveValueSigned() ) {
        long l = obj.GetPrimitiveValueLong();
    } else {
        unsigned long ul = obj.GetPrimitiveValueULong();
    }
    break;
//... etc.
}
```

Member iterator methods are also defined in the **CConstObjectInfo** class, with a similar interface to that found in the **CObjectTypeInfo** class. In this case however, the dereferencing operators return a **CConstObjectInfo** object - not a **CObjectTypeInfo** object - for the current member. For C++*class* objects, these member functions are:

- *CMemberIterator BeginMembers(void) const;*
- *CMemberIterator FindClassMember(const string& memberName) const;*
- *CMemberIterator FindClassMemberByTag(int memberTag) const;*

For C++ *choice* objects, only one variant is ever selected, and only that choice variant is instantiated. As it does not make sense to define a **CConstObjectInfo** *iterator* for uninstantiated variants, the method **GetCurrentChoiceVariant()** is provided instead. The dereferencing operator (\*) can be applied to the object returned by this method to obtain a **CConstObjectInfo** for the variant. Of course, type information for unselected variants can still be accessed using the **CObjectTypeInfo** methods.

The **CConstObjectInfo** class also defines an element iterator for container type objects. **CConstObjectInfo::CElementIterator** is a forward iterator whose interface includes increment and testing operators. Dereferencing is implemented by the iterator's **GetElement()** method, which returns a **CConstObjectInfo** for the element currently pointed to by the iterator.

Finally, for pointer type objects, the type returned by the method **GetPointedObject()** is also a **CConstObjectInfo** for the object - not just a **CObjectTypeInfo**.

## CObjectInfo (%20)

The **CObjectInfo** class is in turn derived from **CConstObjectInfo**, and is intended for usage with *mutable* instances of the object of interest. In addition to all of the methods inherited from the parent class, the interface to this class also provides methods that allow modification of the object itself or its data members.

For primitive type objects, a set of **SetPrimitiveValueXxx()** methods are available, complimentary to the **GetPrimitiveValueXxx()** methods described above. Methods that return member iterator objects are again reimplemented, and the de-referencing operators now return a **CObjectInfo** object for that data member. As the **CObjectInfo** now points to a *mutable* object, these iterators can be used to set values for the data member. Similarly, **GetCurrentChoiceVariant()** now returns a **CObjectInfo**, as does **CObjectInfo::CElementIterator::GetElement()**.

## Usage of object type information

We can now reconsider how our **LoadObject()** function might be implemented using the **CObjectInfo** class:

```
bool LoadObject(CObjectInfo& info, CNcbiIStream& is)
{
    string alias, myValue;

    while ( !is.eof() ) {
        is >> alias >> myValue;

        CObjectInfo dataMember(*info.FindClassMember(alias));
        if (!dataMember) {
            ERR_POST(ERROR, "Couldn't find member named:" << alias);
        }
        SetValue(dataMember, myValue);
    }
}
```

Here, **info** contains pointers to the **CObject** itself as well as to its associated **CTypeInfo** object. For each member alias read from the file, we apply **FindClassMember(alias)**, and dereference the returned iterator to retrieve a **CObjectInfo** object for that member. We then use the operator **()** to verify that the member was located, and if so, use the member's **CObjectInfo** to set a value in the function **SetValue()**:

```
void SetValue(const CObjectInfo& obj, const string value)
{
    if (obj.GetTypeFamily() == eTypeFamilyPrimitive) {

        switch ( obj.GetPrimitiveValueType() ) {

        case ePrimitiveValueBool:
            obj.SetPrimitiveValueBool (atoi (value.c_str()));
            break;
```

```

        case ePrimitiveValueChar:
            obj.SetPrimitiveValueChar (value.c_str()[0]);
            break;

        //... etc
    }
} else {
    ERR_POST(ERROR, "Attempt to assign non-primitive from string:" << value);
}
}

```

In this example, **SetValue()** can only assign primitive types. More generally however, the **CObjectInfo** class allows the assignment of more complex types that are simply not implemented here. Note also that the arguments to **SetValue()** are *const*, even though the function **does** modify the value of the data instance pointed to. In particular, the type *const CObjectInfo* should not be confused with the type **CConstObjectInfo**. The former specifies that object information construct is non-mutable, although the instance it points to can be modified. The latter specifies that the instance itself is non-mutable.

In addition to user-specific applications of the type demonstrated in this example, the generic implementations of the C++ type iterators and the **CObject[IO]Stream** class methods provide excellent examples of how runtime object type information can be deployed.

As a final example of how type information might be used, we consider an application whose simple task is to translate a data file on an input stream to a different format on an output stream. One important use of the object classes defined in *include/objects* is the hooks and parsing mechanisms available to applications utilizing **CObject[IO]Streams**. The stream objects specialize in different formats (such as XML or ASN.1), and must work in concert with these type-specific object classes to interpret or generate serialized data. In some cases however, the dynamic memory allocation required for large objects may be substantial, and it is preferable to avoid actually instantiating a whole object all at once.

Instead, it is possible to use the **CObjectStreamCopier** class, described in **CObject[IO]Streams**. Briefly, this class holds two **CObject[IO]Stream** data members pointing to the input and output streams, and a set of *Copy* methods which take a **CTypeInfo** argument. Using this class, it is easy to translate files between different formats; for example:

```

auto_ptr<CObjectIStream> in(CObjectIStream::Open("mydata.xml", eSerial_Xml));
auto_ptr<CObjectOStream> out(CObjectOStream::Open("mydata.asn", eSerial_AsnBinary));
CObjectStreamCopier copier(*in, *out);
copier.Copy (CBioseq_set::GetTypeInfo());

```

copies a **CBioseq\_set** encoded in XML to a new file, reformatted in ASN.1 binary format.

## Choice objects in the NCBI C++ Toolkit

---

The following topics are discussed in this section:

- Introduction

- C++ choice objects

## Introduction

The `datatool` program processes the ASN.1 specification files (`*.asn`) in the `src/objects/` directories to generate the associated C++ class definitions. The corresponding program implemented in the C Toolkit, `asntool`, used the ASN.1 specifications to generate C enums, structs, and functions. In contrast, `datatool` must generate C++ enums, classes and methods. In addition, for each defined object type, `datatool` must also generate the associated type information method or function.

There is a significant difference in how these two tools implement ASN.1 *choice* elements. As an example, consider the following ASN.1 specification:

```
Object-id ::= CHOICE {
    id INTEGER,
    str VisibleString
}
```

The ASN.1 *choice* element specifies that the corresponding object may be any one of the listed types. In this case, the possible types are an integer and a string. The approach used in `asntool` was to implement all choice objects as **ValNodes**, which were in turn defined as:

```
typedef struct valnode {
    unsigned choice;
    DataVal data;
    struct valnode *next;
} ValNode;
```

The **DataVal** field is a *union*, which may directly store numerical values, or alternatively, hold a **void** pointer to a character string or C *struct*. Thus, to process a *choice* element in the C Toolkit, one could first retrieve the *choice* field to determine how the data should be interpreted, and subsequently, retrieve the data via the **DataVal** field. In particular, no explicit implementation of individual choice objects was used, and it was left to functions which manipulate these elements to enforce logical consistency and error checking for legitimate values. A C *struct* which included a *choice* element as one of its fields merely had to declare that element as type *ValNode*. This design was further complicated by the use of a **void** pointer to store non-primitive types such as *structs* or character strings.

In contrast, the C++ `datatool` implementation of *choice* elements defines a class with built-in, automatic error checking for each *choice* object. The usage of **CObject** class hierarchy (and the associated type information methods) solves many of the problems associated with working with **void** pointers.

## C++ choice objects

The classes generated by `datatool` for *choice* elements all have the following general structure:

```
class C[AsnChoiceName] : public CObject
{
public:
```

```

...                               // constructors and destructors
DECLARE_INTERNAL_TYPE_INFO();     // declare GetTypeInfo() method
enum E_Choice {                   // enumerate the class names
    e_not_set,                    // for the choice variants
    e_Xxx,
    ...
};
typedef CXxx TXxx;                // typedef each variant class
...
virtual void Reset(void);         // reset selection to none
E_Choice Which(void) const;       // return m_choice
void Select(E_Choice index,       // change the current selection
            EResetVariant reset);
static string SelectionName(E_Choice index);
bool IsXxx(void) const;           // true if m_choice == eXxx
CXxx& GetXxx(void);
const CXxx& GetXxx(void) const;
CXxx& SetXxx(void);
void SetXxx(const CRef<CXxx>& ref);
...
private:
    E_Choice m_choice;             // choice state
    union {
        TXxx m_Xxx;
        ...
    };
    CObject *m_object;             // variant's data
    ...
};

```

For the above ASN.1 specification, `datatool` generates a class named ***CObject\_id***, which is derived from ***CObject***. For each choice variant in the specification, an enumerated value (in ***E\_Choice***), and an internal *typedef* are defined, and a declaration in the *union* data member is made. For this example then, we would have:

```

enum E_Choice {
    e_not_set,
    e_Id,
    e_Str
};
...
typedef int TId;
typedef string TStr;
...
union {
    TId m_Id;
    string *m_string;
};

```

In this case both of the choice variants are C++ built-in types. More generally however, the choice variant types may refer to any type of object. For convenience, we refer to their C++ type names here as "CXxx",

Two private data members store information about the currently selected choice variant: `m_choice` holds the *enum* value, and `m_Xxx` holds (or points to a **CObject** containing) the variant's data. The choice object's member functions provide access to these two data members. **Which()** returns the currently selected variant's *E\_Choice enum* value. Each choice variant has its own **Get()** and **Set()** methods. Each **GetXxx()** method throws an exception if the variant type for that method does not correspond to the current selection type. Thus, it is not possible to unknowingly retrieve the incorrect type of choice variant.

**Select(e\_Xxx)** uses a *switch(e\_Xxx)* statement to initialize `m_Xxx` appropriately, sets `m_choice` to `e_Xxx`, and returns. Two **SetXxx()** methods are defined, and both use this **Select()** method. **SetXxx()** with no arguments calls **Select(e\_Xxx)** and returns `m_Xxx` (as initialized by **Select()**). **SetXxx(TXxx& value)** also calls **Select(e\_Xxx)** but resets `m_Xxx` to `value` before returning.

Some example choice objects in the C++ Toolkit are:

- CDate
- CInt\_fuzz
- CObject\_id
- CPerson\_id
- CAnnotdesc
- CSeq\_annot

## Traversing a Data Structure

---

The following topics are discussed in this section:

- Locating the Class Definitions
- Accessing and Referencing Data Members
- Traversing a Biostruc
- Iterating Over Containers

### Locating the Class Definitions

In general, traversing through a class object requires that you first become familiar with the internal class structure and member access functions for that object. In this section we consider how you can access this information in the source files, and apply it. The example provided here involves a **Biostruc** type which is implemented by class **CBiostruc**, and its base (parent) class, **CBiostruc\_Base**.

The first question is: how do I locate the class definitions implementing the object to be traversed? There are now two source browsers which you can use. To obtain a synopsis of the class, you can search the index or the class hierarchy of the *Doc++* browser and follow a link to the class. For example, a synopsis of the **CBiostruc** class is readily available. From this page, you can also access the relevant source files archived by the *LXR* browser, by following the Locate CBiostruc link. Alternatively, you may want to access the *LXR* engine directly by using the Identifier search tool.

Because we wish to determine which headers to include, the synopsis displayed by the Identifier search tool is most useful. There we find a single header file, *Biostruc.hpp*, listed as defining the class. Accordingly, this is the header file we must include. The **CBiostruc** class inherits from the **CBiostruc\_Base** class however, and we will need to consult that file as well to understand the internal structure of the **CBiostruc** class. Following a link to the parent class from the class hierarchy browser, we find the definition of the **CBiostruc\_Base** class.

This is where we must look for the definitions and access functions we will be using. However, it is the *derived user class* (**CBiostruc**) whose header should be *#include'd* in your source files, and which should be instantiated by your local program variable. For a more general discussion of the relationship between the base parent objects and their derived user classes, see Working with the serializable object classes.

## Accessing and Referencing Data Members

Omitting some of the low-level details of the base class, we find the **CBiostruc\_Base** class has essentially the following structure:

```
class CBiostruc_Base : public CObject
{
public:
    // type definitions
    typedef list< CRef<CBiostruc_id> > TId;
    typedef list< CRef<CBiostruc_descr> > TDescr;
    typedef list< CRef<CBiostruc_feature_set> > TFeatures;
    typedef list< CRef<CBiostruc_model> > TModel;
    typedef CBiostruc_graph TChemical_graph;
    // Get() members
    const TId& GetId(void) const;
    const TDescr& GetDescr(void) const;
    const TChemical_graph& GetChemical_graph(void) const;
    const TFeatures& GetFeatures(void) const;
    const TModel& GetModel(void) const;
    // Set() members
    TId& SetId(void);
    TDescr& SetDescr(void);
    TChemical_graph& SetChemical_graph(void);
    TFeatures& SetFeatures(void);
    TModel& SetModel(void);
private:
    TId m_Id;
    TDescr m_Descr;
    TChemical_graph m_Chemical_graph;
```

```
TFeatures m_Features;
TModel m_Model;
};
```

With the exception of the structure's chemical graph, each of the class's private data members is actually a *list* of references (pointers), as specified by the type definitions. For example, **TId** is a list of **CRef** objects, where each **CRef** object points to a **CBiostruc\_id**. The **CRef** class is a type of smart pointer used to hold a pointer to a reference-counted object. The dereferencing operator, when applied to a (dereferenced) iterator pointing to an element of **CBiostruc::TId**, e.g. **\*\*CRef\_i**, will return a **CBiostruc\_id**. Thus, the call to **GetId()** returns a list which must then be iterated over and dereferenced to get the individual **CBiostruc\_id** objects. In contrast, the function **GetChemicalGraph()** returns the object directly, as it does not involve a *list* or a **CRef**.

NOTE: It is strongly recommended that you use type names defined in the generated classes (e.g. **TId**, **TDescr**) rather than generic container names (*list< CRef<CBiostruc\_id> >* etc.). The real container class may change occasionally and you will have to modify the code using generic container types every time it happens. When iterating over a container it's recommended to use **ITERATE** and **NON\_CONST\_ITERATE** macros.

The **GetXxx()** and **SetXxx()** member functions define the user interface to the class, providing methods to access and modify ("mutate") private data. In addition, most classes, including **CBiostruc**, have **IsSetXxx()** and **ResetXxx()** methods to validate and clear the data members, respectively.

## Traversing a Biostruc

The program *traverseBS.cpp* (see Box 4) demonstrates how one might load a serial data file and iterate over the components of the resulting object. This example reads from a text ASN.1 Biostruc file and stores the information into a **CBiostruc** object in memory. The overloaded **Visit()** function is then used to recursively examine the object **CBiostruc bs** and its components.

**Visit(bs)** simply calls **Visit()** on each of the **CBiostruc** data members, which are accessed using **bs.GetXxx()**. The information needed to write each of these functions - the data member types and member function signatures - is contained in the respective header files. For example, looking at *Biostruc\_.hpp*, we learn that the structure's descriptor list can be accessed using **GetDescr()**, and that the type returned is a list of pointers to descriptors:

```
typedef list< CRef<CBiostruc_descr> > TDescr;
const TDescr& GetDescr(void) const;
```

Consulting the base class for **CBiostruc\_descr** in turn, we learn that this class has a *choice state* defining the type of value stored there as well as the method that should be used to access that value. This leads to an implementation of **Visit(CBiostruc::TDescr DescrList)** that uses an iterator over its list argument and a switch statement over the current descriptor's choice state.

## Iterating Over Containers

Most of the **Visit()** functions implemented here rely on standard STL iterators to walk through a list of objects. The general syntax for using an iterator is:

```

ContainerType ContainerName;
for (ContainerType::IteratorType
    i = ContainerName.begin(); i != ContainerName.end(); ++i) {

    ObjectType ObjectName = *i;
    // ...
}

```

Dereferencing the iterator is required, as the iterator behaves like a pointer that traverses consecutive elements of the container. For example, to iterate over the list of descriptors in the *Biostruc*, we use a container of type **CBiostruc::TDescr**, and an iterator of type *const\_iterator* to ensure that the data is not mutated in the body of the loop. Because the descriptor list contains pointers (**CRefs**) to objects, we will actually need to dereference **twice** to get to the objects themselves.

```

for (CBiostruc::TDescr::const_iterator i = descList.begin();
    i != descList.end(); ++i) {

    const CBiostruc_descr& thisDescr = **i;
    // ...
}

```

In traversing the descriptor list in this example, we handled each type of descriptor with an explicit case statement. In fact, however, we really only visit those descriptors whose types have string representations: **TName**, **TPdb\_comment**, and **TOther\_comment**. The other two descriptor types, **THistory** and **TAttribute**, are objects that are "visited" recursively, but the associated visit functions are not actually implemented (see Box 5, *traverseBS.hpp*).

The NCBI C++ Toolkit provides a rich and powerful set of iterators for various application needs. An alternative to using the above *switch* statement to visit elements of the descriptor list would have been to use an NCBI **CStdTypeIterator** that only visits strings. For example, we could implement the Visit function on a **CBiostruc::TDescr** as follows:

```

void Visit (const CBiostruc::TDescr& descList)
{
    for (CBiostruc::TDescr::const_iterator i1 = descList.begin();
        i1 != descList.end(); ++i1) {

        for (CStdTypeConstIterator<string> i = ConstBegin(**i1); i; ++i) {
            cout << *i << endl;
        }
    }
}

```

In this example, the iterator will skip over all but the string data members.

The **CStdTypeIterator** is one of several iterators which makes use of an object's *type information* to implement the desired functionality. We began this section by positing that the traversal of an object requires an a priori knowledge of that object's internal structure. This is not strictly true however, if type information for the object is also available. An object's type information specifies the class layout, inheritance relations, data member names, and various other attributes

such as size, which are independent of specific instances. All of the C++ type iterators described in The NCBI C++ Toolkit Iterators section utilize type information, which is the topic of the next section: Runtime Object Type Information.

## Box 1: xml2asn.cpp

```
// File name: xml2asn.cpp
// Description: Reads an XML Biostruc file into memory
//             and saves it in ASN.1 text and binary formats.
#include <corelib/ncbiapp.hpp>
#include <serial/serial.hpp>
#include <serial/objistr.hpp>
#include <serial/objostr.hpp>
#include <objects/mmdbl/Biostruc.hpp>
USING_NCBI_SCOPE;
class CTestAsn : public CNcbiApplication {
public:
    virtual int Run ();
};
using namespace objects;
int CTestAsn::Run() {
    auto_ptr<CObjectIStream>
        xml_in(CObjectIStream::Open("1001.xml", eSerial_Xml));
    auto_ptr<CObjectOStream>
        txt_out(CObjectOStream::Open("1001.asntxt", eSerial_AsnText));
    auto_ptr<CObjectOStream>
        bin_out(CObjectOStream::Open("1001.asnbin", eSerial_AsnBinary));
    CBiostruc bs;
    *xml_in >> bs;
    *txt_out << bs;
    *bin_out << bs;
    return 0;
}
int main(int argc, const char* argv[])
{
    CNcbiOfstream diag("asntrans.log");
    SetDiagStream(&diag);
    CTestAsn theTestApp;
    return theTestApp.AppMain(argc, argv);
}
```

## Box 2: ctypeiter.cpp

```

// File name: ctypeiter.cpp
// Description: Demonstrate using a CTypeIterator
// Notes: build with xncbi and xser libraries#include "ctypeiter.hpp"#include <serial/serial.hpp>
#include <serial/objistr.hpp>
#include <serial/objostr.hpp>
#include <serial/iterator.hpp>
#include <serial/serialimpl.hpp> // type information for class CPersonBEGIN_CLASS_INFO(CPerson){
    ADD_STD_MEMBER(m_Name);
    ADD_STD_MEMBER(m_Addr);
    ADD_MEMBER(m_NextDoor, POINTER, (CLASS, (CPerson)))>SetOptional();
}END_CLASS_INFO// type information for class CDistrictBEGIN_CLASS_INFO(CDistrict){
    ADD_STD_MEMBER(m_Number);
    ADD_MEMBER(m_Blocks, STL_list, (CLASS, (CPerson)));
}END_CLASS_INFO// main and other functionsUSING_NCBI_SCOPE;static void FullerBrushMan (const
CPerson& p) {
    cout << "knock-knock! is " << p.m_Name << " home?" << endl;
}
int main(int argc, char** argv)
{
    // Instantiate a few CPerson objects    CPerson neighborhood("Moe", "1 Main St",
        new CPerson("Larry", "2 Main St",
            new CPerson("Curly", "3 Main St", 0));
    CPerson another ("Harpo", "2 River Rd",
        new CPerson("Chico", "4 River Rd",
            new CPerson("Groucho", "6 River Rd", 0));

    // Create a CDistrict and install some CPerson objects    CDistrict district1(1);
    district1.AddBlock(neighborhood);
    district1.AddBlock(another);
    // Send the FullerBrushMan to all CPersons in district1
    for (CTypeConstIterator<CPerson> house = ConstBegin(district1);
        house; ++house ) {
        FullerBrushMan(*house);
    }
    // Iterate over all strings for the CPersons in district1
    list<CPerson> blocks = district1.GetBlocks();
    for (list<CPerson>::iterator b = blocks.begin();
        b != blocks.end(); ++b) {
        for (CStdTypeIterator<string> i = Begin(*b); i; ++i) {
            cout << *i << ' ';
        }
        cout << endl;
    }
    return 0;
}

```

## Box 3: ctypeiter.hpp

```

// File name: ctypeiter.hpp#ifndef CTYPEITER_HPP
#define CTYPEITER_HPP
#include <corelib/ncbistd.hpp>
#include <corelib/ncbiobj.hpp>
#include <serial/typeinfo.hpp>
#include <string>
#include <list>USING_NCBI_SCOPE;class CPerson
{
public:    CPerson(void)
        : m_Name(0), m_Addr(0), m_NextDoor(0)
        {}
    CPerson(string n, string s, CPerson* p)
        : m_Name(n), m_Addr(s), m_NextDoor(p)
        {}
    virtual ~CPerson(void) {}
    static const CTypeInfo* GetTypeInfo(void);
    string m_Name, m_Addr;
    CPerson *m_NextDoor;
};
class CDistrict
{
public:    CDistrict(void) : m_Number(0) {}
        CDistrict(int n) : m_Number(n) {}
        virtual ~CDistrict(void) {}
        static const CTypeInfo* GetTypeInfo(void);
        int m_Number;
        void AddBlock (const CPerson& p) { m_Blocks.push_back(p); }
        list<CPerson>& GetBlocks() { return m_Blocks; }
private:    list<CPerson> m_Blocks;
};
#endif /* CTYPEITER_HPP */

```

## Box 4: traverseBS.cpp

```

// File name: traverseBS.cpp
// Description: Reads an ASN.1 Biostruc text file into memory
// and visits its components#include <serial/serial.hpp>
#include <serial/iterator.hpp>
#include <serial/objistr.hpp>
#include <serial/serial.hpp>
#include <objects/general/Dbtag.hpp>
#include <objects/general/Object_id.hpp>
#include <objects/seq/Numbering.hpp>
#include <objects/seq/Pubdesc.hpp>
#include <objects/seq/Heterogen.hpp>
#include <objects/mmdb1/Biostruc.hpp>
#include <objects/mmdb1/Biostruc_id.hpp>
#include <objects/mmdb1/Biostruc_history.hpp>
#include <objects/mmdb1/Mmdb_id.hpp>
#include <objects/mmdb1/Biostruc_descr.hpp>
#include <objects/mmdb1/Biomol_descr.hpp>
#include <objects/mmdb1/Molecule_graph.hpp>
#include <objects/mmdb1/Inter_residue_bond.hpp>
#include <objects/mmdb1/Residue_graph.hpp>
#include <objects/mmdb3/Biostruc_feature_set.hpp>
#include <objects/mmdb2/Biostruc_model.hpp>
#include <objects/pub/Pub.hpp>
#include <corelib/ncbistre.hpp>
#include "traverseBS.hpp"USING_NCBI_SCOPE;
using namespace objects;int CTestAsn::Run()
{
    // initialize ASN input stream
    auto_ptr<CObjectIStream>
        inObject(CObjectIStream::Open("1001.val", eSerial_AsnBinary));
    // initialize, read into, and traverse CBiostruc object      CBiostruc bs;
    *inObject >> bs;
    Visit (bs);

    return 0;
}
/*****
*
* The overloaded free "visit" functions are used to explore the
* Biostruc and all its component members - most of which are also
* class objects. Each class has a public interface that provides
* access to its private data via "get" functions.
*
*****/void Visit (const CBiostruc&
bs)
{
    cout << "Biostruc:\n" << endl;
    Visit (bs.GetId());
    Visit (bs.GetDescr());
    Visit (bs.GetChemical_graph());

```

```

    Visit (bs.GetFeatures());
    Visit (bs.GetModel());
}

/*****
*
* TId is a type defined in the CBiostruc class as a list of CBiostruc_id,
* where each id has a choice state and a value. Depending on the choice
* state, a different get() function is used.
*
*****/
void Visit (const CBiostruc::TId& idList)
{
    cout << "\n Visiting Ids of Biostruc:\n";

    for (CBiostruc::TId::const_iterator i = idList.begin();
        i != idList.end(); ++i) {

        // dereference the iterator to get to the id object          const CBiostruc_id& thisId =
**i;

        CBiostruc_id::E_Choice choice = thisId.Which();
        cout << "choice = " << choice;

        // select id's get member function depending on choice      switch (choice) {
        case CBiostruc_id::e_Mmdb_id:
            cout << " mmdbId: " << thisId.GetMmdb_id().Get() << endl;
            break;
        case CBiostruc_id::e_Local_id:
            cout << " Local Id: " << thisId.GetLocal_id().GetId() << endl;
            break;
        case CBiostruc_id::e_Other_database:
            cout << " Other DB Id: "
                << thisId.GetOther_database().GetDb() << endl;
            break;
        default:
            cout << "Choice not set or unrecognized" << endl;
        }
    }
}

/*****
*
* TDescr is also a type defined in the Biostruc class as a list of
* CBiostruc_descr, where each descriptor has a choice state and a value.
*
*****/
void Visit (const CBiostruc::TDescr& descList)
{
    cout << "\n Visiting Descriptors of Biostruc:\n";

    for (CBiostruc::TDescr::const_iterator i = descList.begin();
        i != descList.end(); ++i) {

```

```

        // dereference the iterator to get the descriptor          const CBiostruc_descr&
thisDescr = **i;
    CBiostruc_descr::E_Choice choice = thisDescr.Which();
    cout << "choice = " << choice;

    // select the get function depending on choice
    switch (choice) {
    case CBiostruc_descr::e_Name:
        cout << " Name: " << thisDescr.GetName() << endl;
        break;
    case CBiostruc_descr::e_Pdb_comment:
        cout << " Pdb comment: " << thisDescr.GetPdb_comment() << endl;
        break;
    case CBiostruc_descr::e_Other_comment:
        cout << " Other comment: " << thisDescr.GetOther_comment() << endl;
        break;
    case CBiostruc_descr::e_History:
        cout << " History: " << endl;
        Visit (thisDescr.GetHistory());
        break;
    case CBiostruc_descr::e_Attribution:
        cout << " Attribute: " << endl;
        Visit (thisDescr.GetAttribution());
        break;
    default:
        cout << "Choice not set or unrecognized" << endl;
    }
    }
    VisitWithIterator (descList);
}

/*****
 *
 * An alternate way to visit the descriptor nodes using a CStdStringIterator
 *
 *****/void
VisitWithIterator (const CBiostruc::TDescr& descList) {
    cout << "\n Revisiting descriptor list with string iterator...\n";

    for (CBiostruc::TDescr::const_iterator i1 = descList.begin();
        i1 != descList.end(); ++i1) {

        const CBiostruc_descr& thisDescr = **i1;

        for (CStdStringConstIterator<NCBI_NS_STD::string>
            i = ConstBegin(thisDescr); i; ++i) {
            cout << "next descriptor" << *i << endl;
        }
    }
}

/*****
 *
 * Chemical graphs contain lists of descriptors, molecule_graphs, bonds, and
 * residue graphs. Here we just visit some of the descriptors.
 *****/

```

```

*
*****/void Visit (const
CBiostruc::TChemical_graph& G)
{
    cout << "\n\n Visiting Chemical Graph of Biostruc\n";

    const CBiostruc_graph::TDescr& descList = G.GetDescr();
    for (CBiostruc_graph::TDescr::const_iterator i = descList.begin();
        i != descList.end(); ++i) {

        // dereference the iterator to get the descriptor          const CBiomol_descr& thisDescr
= **i;
        CBiomol_descr::E_Choice choice = thisDescr.Which();
        cout << "choice = " << choice;

        // select the get function depending on choice
        switch (choice) {
        case CBiomol_descr::e_Name:
            cout << " Name: " << thisDescr.GetName() << endl;
            break;
        case CBiomol_descr::e_Pdb_class:
            cout << " Pdb class: " << thisDescr.GetPdb_class() << endl;
            break;
        case CBiomol_descr::e_Pdb_source:
            cout << " Pdb Source: " << thisDescr.GetPdb_source() << endl;
            break;
        case CBiomol_descr::e_Pdb_comment:
            cout << " Pdb comment: " << thisDescr.GetPdb_comment() << endl;
            break;
        case CBiomol_descr::e_Other_comment:
            cout << " Other comment: " << thisDescr.GetOther_comment() << endl;
            break;
        case CBiomol_descr::e_Organism: // skipped
        case CBiomol_descr::e_Attribution:
            break;
        case CBiomol_descr::e_Assembly_type:
            cout << " Assembly Type: " << thisDescr.GetAssembly_type() << endl;
            break;
        case CBiomol_descr::e_Molecule_type:
            cout << " Molecule Type: " << thisDescr.GetMolecule_type() << endl;
            break;
        default:
            cout << "Choice not set or unrecognized" << endl;
        }
    }
}

void Visit (const CBiostruc::TFeatures&)
{
    cout << "\n\n Visiting Features of Biostruc\n";
}

void Visit (const CBiostruc::TModel&)
{

```

```
    cout << "\n\n Visiting Models of Biostruc\n";
}
int main(int argc, const char* argv[])
{
    // initialize diagnostic stream    CNcbiOfstream diag("traverseBS.log");
    SetDiagStream(&diag);

    CTestAsn theTestApp;
    return theTestApp.AppMain(argc, argv);
}
```

## Box 5: traverseBS.hpp

```
// File name traverseBS.hpp#ifndef NCBI_TRAVERSEBS_HPP
#define NCBI_TRAVERSEBS_HPP
#include <corelib/ncbistd.hpp>
#include <corelib/ncbiapp.hpp>
USING_NCBI_SCOPE;
using namespace objects;// class CTestAsn
class CTestAsn : public CNcbiApplication {
public:
    virtual int Run ();
};
void Visit(const CBiostruc&);
void Visit(const CBiostruc::TId&);
void Visit(const CBiostruc::TDescr&);
void Visit(const CBiostruc::TChemical_graph&);
void Visit(const CBiostruc::TFeatures&);
void Visit(const CBiostruc::TModel&);
void Visit(const CBiostruc_history&) {
    cout << "visiting history" << endl;
};
// Not implemented
void Visit(const CBiostruc_descr::TAttribution&) {};
void VisitWithIterator (const CBiostruc::TDescr& descList);
#endif /* NCBI_TRAVERSEBS_HPP */
```